

# User Manual for Rodin v.2.2

Work in Progress  
Handbook Rev: 12578

[rodin-handbook@formalmind.com](mailto:rodin-handbook@formalmind.com)  
[handbook.event-b.org](http://handbook.event-b.org)

August 3, 2011



# Contents

|          |                                               |           |
|----------|-----------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                           | <b>7</b>  |
| 1.1      | Overview                                      | 7         |
| 1.2      | Foreword                                      | 7         |
| 1.3      | Conventions                                   | 7         |
| 1.4      | Acknowledgements                              | 8         |
| 1.5      | Creative Commons License                      | 8         |
| 1.6      | Style Guide                                   | 8         |
| <b>2</b> | <b>Tutorial</b>                               | <b>11</b> |
| 2.1      | Outline                                       | 11        |
| 2.2      | Before Getting Started                        | 12        |
| 2.2.1    | Systems Development                           | 12        |
| 2.2.2    | Formal Modelling                              | 12        |
| 2.2.3    | Predicate Logic                               | 12        |
| 2.2.4    | Event-B                                       | 13        |
| 2.2.5    | Rodin                                         | 13        |
| 2.2.6    | Eclipse                                       | 13        |
| 2.3      | Installation                                  | 13        |
| 2.3.1    | Install Rodin for the first time              | 13        |
| 2.3.2    | Install new plugins                           | 14        |
| 2.4      | The First Machine: A Traffic Light Controller | 15        |
| 2.4.1    | Project Setup                                 | 15        |
| 2.4.2    | Building the Model                            | 18        |
| 2.4.3    | The Final Traffic Light Model                 | 20        |
| 2.5      | Mathematical notation                         | 21        |
| 2.5.1    | Predicates                                    | 21        |
| 2.5.2    | Data types                                    | 22        |
| 2.5.3    | Operations on Sets                            | 22        |
| 2.5.4    | Introducing user-defined types                | 23        |
| 2.5.5    | Relations                                     | 23        |
| 2.5.6    | Arithmetic                                    | 24        |
| 2.6      | Introducing Contexts                          | 24        |
| 2.6.1    | Create a Context                              | 24        |
| 2.6.2    | Populate the Context                          | 24        |
| 2.6.3    | The Final Context                             | 27        |
| 2.7      | Event-B Concepts                              | 28        |
| 2.7.1    | Contexts                                      | 28        |
| 2.7.2    | Machines                                      | 29        |
| 2.7.3    | Events                                        | 29        |
| 2.7.4    | Refinement                                    | 30        |

|          |                                                             |           |
|----------|-------------------------------------------------------------|-----------|
| 2.8      | Expanding the Traffic Light System: Contexts and Refinement | 31        |
| 2.8.1    | Data Refinement                                             | 31        |
| 2.8.2    | A Context with Colors                                       | 32        |
| 2.8.3    | The Actual Data Refinement                                  | 32        |
| 2.8.4    | The refined machine with data refinement for peds-go        | 33        |
| 2.8.5    | Witnesses                                                   | 35        |
| 2.8.6    | Discussion                                                  | 36        |
| 2.8.7    | The Refined Machine with All Data Refinement                | 36        |
| 2.8.8    | One more Refinement: The Push Button                        | 37        |
| 2.8.9    | Discussion                                                  | 38        |
| 2.9      | Proving                                                     | 39        |
| 2.9.1    | The Celebrity Problem                                       | 39        |
| 2.9.2    | The Final Second Refinement                                 | 40        |
| 2.9.3    | The First Proof                                             | 41        |
| 2.10     | Location Access Controller                                  | 44        |
| 2.10.1   | Initial Model                                               | 44        |
| 2.10.2   | First Refinement                                            | 47        |
| 2.11     | Outlook                                                     | 49        |
| <b>3</b> | <b>Reference</b>                                            | <b>51</b> |
| 3.1      | Outline                                                     | 51        |
| 3.2      | Abstract Machine                                            | 52        |
| 3.3      | Abstract Machine Notation                                   | 52        |
| 3.4      | Arithmetic                                                  | 52        |
| 3.5      | Atelier B Provers                                           | 52        |
| 3.6      | Auto Prover                                                 | 52        |
| 3.7      | Axiom                                                       | 52        |
| 3.8      | B                                                           | 52        |
| 3.9      | Camille                                                     | 53        |
| 3.10     | Classical B                                                 | 53        |
| 3.11     | Constant                                                    | 53        |
| 3.12     | Context                                                     | 53        |
| 3.13     | Data Refinement                                             | 53        |
| 3.14     | Datatypes                                                   | 53        |
| 3.15     | Deadlock                                                    | 53        |
| 3.16     | Deploy                                                      | 53        |
| 3.17     | Eclipse                                                     | 53        |
| 3.18     | Editor View                                                 | 53        |
| 3.19     | Event                                                       | 53        |
| 3.20     | Event-B                                                     | 53        |
| 3.21     | Event-B Component                                           | 54        |
| 3.22     | Event-B Explorer                                            | 54        |
| 3.23     | First Order Predicate Calculus                              | 54        |
| 3.24     | Formal Method                                               | 54        |
| 3.25     | Gluings Invariant                                           | 54        |
| 3.26     | Goal View                                                   | 54        |
| 3.27     | Guard                                                       | 54        |
| 3.28     | IDE                                                         | 54        |
| 3.29     | Initialization                                              | 54        |
| 3.30     | Label                                                       | 54        |
| 3.31     | Machine                                                     | 54        |
| 3.32     | Mathematical Notation                                       | 54        |

|                                                                |           |
|----------------------------------------------------------------|-----------|
| 3.33 Menu Bar                                                  | 54        |
| 3.34 Model Checker                                             | 54        |
| 3.35 Naming Convention                                         | 54        |
| 3.36 Non-Deterministic                                         | 55        |
| 3.37 Outline View                                              | 55        |
| 3.38 P0 Prover                                                 | 55        |
| 3.39 Partition                                                 | 55        |
| 3.40 Proof Control View                                        | 55        |
| 3.41 Proof Obligation                                          | 55        |
| 3.42 Proof Tree View                                           | 55        |
| 3.43 Plugin                                                    | 55        |
| 3.44 Refinement                                                | 55        |
| 3.45 Rodin Wiki                                                | 55        |
| 3.46 Structural Editor                                         | 56        |
| 3.47 Symbols View                                              | 56        |
| 3.48 Predicate Logic                                           | 56        |
| 3.49 ProB                                                      | 56        |
| 3.50 Project                                                   | 56        |
| 3.51 Proof Obligation Labels                                   | 56        |
| 3.52 Propositional Calculus                                    | 56        |
| 3.53 Relation                                                  | 56        |
| 3.54 Rodin                                                     | 56        |
| 3.55 Rodin Download                                            | 56        |
| 3.56 Rodin Platform                                            | 56        |
| 3.57 Rodin Problems View                                       | 56        |
| 3.58 Rodin Nature                                              | 56        |
| 3.59 Sees                                                      | 56        |
| 3.60 Set                                                       | 56        |
| 3.61 Set Theory                                                | 56        |
| 3.62 Temporal Logic                                            | 57        |
| 3.63 Theorem                                                   | 57        |
| 3.64 Tool Bar                                                  | 57        |
| 3.65 Undischarged Proof Obligations                            | 57        |
| 3.66 Upgrade                                                   | 57        |
| 3.67 Witness                                                   | 57        |
| 3.68 Wizard                                                    | 57        |
| 3.69 Zip File                                                  | 57        |
| <b>4 Frequently Asked Questions</b>                            | <b>59</b> |
| 4.1 General Questions                                          | 59        |
| 4.1.1 What is Event-B?                                         | 59        |
| 4.1.2 What is the difference between Event-B and the B method? | 59        |
| 4.1.3 What is Rodin?                                           | 59        |
| 4.1.4 Where does the Rodin name come from?                     | 59        |
| 4.2 Installation Questions                                     | 60        |
| 4.3 Proofer Questions                                          | 60        |
| 4.4 Usage Questions                                            | 60        |
| 4.4.1 Where vs. When: What's going on?                         | 60        |
| 4.5 Plug-In Questions                                          | 60        |



# Chapter 1

## Introduction

The Rodin (3.56) documentation needs improvement. This has been established at the Deploy Exec Meeting in 2010. We've been collecting background information and the requirements in the [Rodin Wiki](#).

### 1.1 Overview

We decided on the following approach:

- The documentation is kept as L<sup>A</sup>T<sub>E</sub>X.
- We keep the source in Subversion. That way anybody with access to Subversion can contribute.
- By keeping the documentation in Subversion, we can keep Rodin and its documentation in sync.
- We use [plasTeX](#) to generate HTML and Eclipse Help from the L<sup>A</sup>T<sub>E</sub>X. plasTeX supports templates, so we can use different templates for the two versions (for instance, we would not generate navigation elements for Eclipse Help, as Eclipse Help itself provides navigation).

This document demonstrates this approach with a very small portion of the existing Rodin documentation. Obviously, it is just a proof of concept.

### 1.2 Foreword

It would be nice to recruit somebody for the foreword - maybe Cliff and/or Jean-Raymond.

### 1.3 Conventions

We use the following conventions in this manual:



Checklists and Milestones are designated with tick. Here we summarize what we want to learn or should have learned so far.



Useful information and tricks are designated by the information sign.



Potential problems and warnings are designated by a warning sign.



Examples and Code are designated by a pencil.

We use **typewriter** font for file names and directories.

We use **sans serif** font for GUI elements like menus and buttons. Menu actions are depicted by a chain of elements, separated by “**>**”, e.g. File **>** New **>** Event-B Component.

## 1.4 Acknowledgements

Icons from Pixel-Mixer<sup>1</sup> – thanks!

## 1.5 Creative Commons License

## 1.6 Style Guide



For now, we will manage the style guide as L<sup>A</sup>T<sub>E</sub>X together with the rest of the documentation. We may take it out upon publication.

### General Stylistic Guidelines

- The Conventions (1.3) are part of the style guide.
- Use the “we” form.
- We use British English.

### Files

Files should be saved in the `files` subdirectory. You can also create a subdirectory. Then, use the definition

```
\file{1}{2}
```

to create a link to the file. The definition takes two arguments. The first argument (1) defines the file name i.e. `Celebrity.zip` or in case if you saved the file in a subdirectory `subdirecotry/Celebrity.zip`. The second argument (2) defines the name which should be displayed in the document, i.e. “Celebrity Problem Example Download”.



Please note, that you only enter the file name without a path before (expected subdirectories). The build script assigns automatically the correct path to the file on the server.

Here is an example using the definition: [Celebrity Problem Example Download](#) .

### Avoiding Redundancy

We will reduce (or avoid) redundancy through heavy linking, following these guidelines:

- If in doubt, provide the bulk of the information in the Reference section. For instance, the FAQ entry “What is Event-B?” Should simply refer to the Event-B entry in the Reference section.
- Web Links should not appear multiple times
- Realize web links as footnotes in the Tutorial and FAQ.
- Realize web links in a “See also” Section in the Reference.

---

<sup>1</sup><http://pixel-mixer.com/>

## Sections

- If referring to a specific chapter or section, use uppercase to denote this, e.g. “in Chapter 3”.
- We also refer to subsections as Section, e.g. “see Section 2.5”
- We have a small number of well-defined chapters which are the top level structuring element.
- Sections and subsections are numbered. In the HTML-Versions, they are broken into subpages.
- Subsubsections do not receive numbers and are not broken into subpages in the HTML. Keep this in mind regarding both the reading flow and page sizes.
- Avoid linking (ref) to subsubsections, as they don’t have a number. Latex will instead provide a link to the next higher element. This works, but it could create confusion.
- Generally, we should avoid gaps in the hierarchy (i.e. having a subsubsection in a section without a subsection in between).<sup>2</sup>
- Section labels should be all in lower case. Use “\_” for blanks.
- We use the prefix “**int\_**” for introduction section labels, “**tut\_**” for tutorial section labels following the section number (i.e. **tut\_01**) and “**faq\_**” for faq section labels following a short version of the title (i.e. **faq.diff.eventb\_b**). Reference section labels have no prefix.

## Images

- Images must be no more than 700 pixels in width (for HTML version) This is fairly easy for bitmaps (screenshot), pay attention to this regarding how plasTeX converts vector images. (see Latex section below on how to include images)
- Screenshots should look neat and consistent. Horizontal real estate will always be an issue, so please resize the windows before taking the screenshot to keep things readable at 700 pixel width. (see Latex section below on how to include images)
- Images should always have a caption and a label. Please use for the label the same conventions like for the section labels. However, use the prefix “**fig\_**”. Example: “**fig\_tut\_03\_traffic\_light**”. For referencing do not use references like “As shown bellow” or “Like in the following image:”. Use always a format like:

“... as shown in figure \ref{fig\_tut\_03\_traffic\_light}.”

- We use icons from Pixel-Mixer, which are free as long as credit is given: <http://www.softicons.com/free-icons/toolbar-icons/basic-icons-by-pixelmixer>
- We include Window decoration only when it is really necessary. If we discuss only some views, we crop the rest away. Please crop neatly, following edges. Even if you need a screenshot with window decoration, you should always use the same Window decoration (i.e. linux ubuntu default decoration style). If you need such a screenshot, please contact Lukas.
- Image file names should be all in lower case and not include umlaute or special characters. Use “\_” for blanks.
- Image files should keep the following rules:

---

<sup>2</sup>Coincidentally, this style guide violates this rule. Reason: We want the style guide not broken into subpages, but the proper hierarchy is a section.

- Tutorial images should be saved in the sub folder `img/tutorial` with the prefix “`tut_`” following the section number. For instance, `tut_01.image1.png`.
- FAQ images should be saved in the sub folder `img/faq` with the prefix “`faq-`”. For instance, `faq_image1.png`.

### **L<sup>A</sup>T<sub>E</sub>X Styling**

- Try to avoid fancy L<sup>A</sup>T<sub>E</sub>X formatting, as PlasTeX (used for generating HTML) is temperamental. Especially macros don’t always work, and sometimes the result is just ugly.
- We have the option to use different files for PDF and HTML generation, but we would generally prefer not to do this. Look at `bsymb.sty` and `plastex-bsymb.sty` as an example. **NOTE:** We don’t do this any more for any style files.
- Every section should have a label, reflecting the section name, all lowercase, spaces replaced with underscores (`_`).
- Don’t create subdirectories in the `latex` folder, as the scripts cannot always deal with them.
- Put images in the `img` folder. Feel free to create additional directory structures underneath.
- Files other than images (e.g. Event-B projects) TODO - we have to figure out whether to keep them in svn (then they won’t be accessible from PDFs) or on the Wiki (then they won’t be accessible offline).
- Don’t use hyperlinks for cross-references, but linked section numbers (generated with `ref`). This is necessary for the print documentation to be useful.
- When including images in Latex, do not provide a width! Instead, try to embed the print size in the image itself. For instance, PNGs allow you to set the print size (in mm). This way we can be sure that the images are rendered as HTML without distortion.

# Chapter 2

## Tutorial

The objective is to get you to a stage where you can use Rodin and build Event-B models. We expect you to have a basic understanding of logic and an idea why doing formal modeling is a good idea. You should be able to work through the tutorial with no or little outside help.

This tutorial covers installation and configuration of Rodin; it brings you through step by step through building formal models and it provides the essential theory and provides pointers to more information.

We attempt to alternate between theory and practical application thereby keeping you motivated. We encourage you not to download solutions to the examples but instead to actively build them up yourself as the tutorial progresses.

If something is unclear, remember to check the Reference (3) for more information.

### 2.1 Outline

**Background before getting started (2.2)** We give a brief description of what Event-B is and what it is being used for and what kind of background knowledge we expect.

**Installation (2.3)** We guide you through downloading, installing and starting Rodin and point out platform differences. We install the provers. We name the visible views and describe what they are doing.

**A Machine, and nothing else (2.4)** We introduce a first machine: a traffic light with booleans for signals. We introduce guards, resulting in the proof obligations to be discharged automatically. We explain how proof labels are read, without changing to the proof perspective.

**Mathematical notation (2.5)** At this point we quickly go through the most important aspects of predicate calculus and provide pointers to the reference and to external literature. We cover everything used by the traffic light system, we introduce all data types and we introduce sets and relations but not in depth. Difference between predicates and expressions; for instance, understand the difference between TRUE and  $\top$ .

**Introducing Contexts (2.6)** We introduce Contexts to apply the theoretical concepts that were introduced in the previous section. We use the Agatha-Puzzle to step by step introduce more and more complex elements. We point out partitions as a typical pitfall (also add to FAQ). We will cover Theorems. Well-Definedness is mentioned.

**Event-B Concepts (2.7)** This is another theoretical section that provides more background about the previous examples. For instance, we analyze the anatomy of a machine and introduce all elements that a machine or context may have. We point to literature about the theory but won't go into the details of the calculus. We describe the sees and refines concepts which will be applied in the next section. We will briefly mention concepts like data refinement and witnesses but leave the details to the literature.

**Expanding the Traffic Light System (2.8)** We apply what we learned in the previous section by introducing a context with traffic light colors and a refinement to integrate them. We will introduce another refinement for the push buttons.

**Proving (2.9)** So far all proof obligations were discharged automatically. Now we switch for the first time to the proving perspective and explore it. We change the auto prover configuration, invalidate proofs and show, that with the new configuration they don't discharge any more. We prove a simple proof by hand. We describe the provers available.

**Complete Abrial Example (2.10)** We pick an interesting example from the Abrial book, if we get permission. We could also take one of the Rodin Wiki Tutorial examples (e.g. Location Access Controller).

**Outlook (2.11)** This concludes the tutorial, but we will provide many pointers for further reading. In particular, we will point to the literature from the Deploy project, the Wiki and to plugins that solve specific problems.

## 2.2 Before Getting Started

Before we get started with the actual tutorial, we are going to go over the required background to make sure that you have a rudimentary understanding of the necessary concepts.

 **You can skip this section, if...**

- ... you know what formal modelling is
- ... you know what predicate logic is
- ... you know what Event-B and Rodin refer to
- ... you know what Eclipse is

### 2.2.1 Systems Development

Ultimately, the purpose of the methods and tools introduced here is to improve systems development. By this we mean the design and management of complex engineering projects over their life cycle. Examples include cars, air traffic control systems, etc.

“Taking an interdisciplinary approach to engineering systems is inherently complex since the behaviour of and interaction among system components is not always immediately well defined or understood. Defining and characterizing such systems and subsystems and the interactions among them is one of the goals of systems engineering. In doing so, the gap that exists between informal requirements from users, operators, marketing organizations, and technical specifications is successfully bridged.”<sup>1</sup>

### 2.2.2 Formal Modelling

We are concerned with *formalizing specifications*. This allows us a more rigorous analysis (thereby improving the quality) and allows the reuse of the specification to develop an implementation. This comes at the cost of higher up-front investments.

This differs from the traditional development process. In a formal development, we transfer some effort from the test phase (where the implementation is verified) to the specification phase (where the specification in relation to the requirements is verified).

### 2.2.3 Predicate Logic

Predicate logic (3.48) is a mathematical logic containing variables that can be quantified.

Event-B supports first-order logic (3.23) which is, technically speaking, just one type of predicate logic.

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Systems\\_engineering#Managing\\_complexity](http://en.wikipedia.org/wiki/Systems_engineering#Managing_complexity)

### 2.2.4 Event-B

Event-B (3.20) is a notation for formal modelling based around an abstract machine notation (3.3).

Event-B is considered an evolution of B (also known as classical B (3.10)). It is a simpler notation which is easier to learn and use. It comes with tool support in the form of the Rodin Platform.

### 2.2.5 Rodin

Rodin (3.56) is the name of the tool platform for Event-B. It allows formal Event-B models to be created with an editor. It generates proof obligations (3.41) that can be discharged either automatically or interactively.

Rodin is modular software and many extensions are available. These include alternative editors, document generators, team support, and extensions (called plugins) to the notation which include decomposition or record support. An up-to-date list of plugins is maintained in the Rodin Wiki (3.45).

### 2.2.6 Eclipse

Rodin is based on the Eclipse Platform (3.17), a Java-based platform for building software tools. This matters for two reasons:

- If you have already used Eclipse-based software, then you will feel immediately comfortable with the handling of the Rodin application.
- Many extensions, or plugins (3.43), are available for Eclipse-based software. There are Rodin-specific plugins as well as Rodin-independent plugins that may be useful to you. The Rodin Wiki (3.45), contains a list of plugins is maintained.

The GUI of an Eclipse application consists of Views, Editors, Toolbars, Quickview, Perspectives and many more elements. If these terms are unfamiliar to you, please consult Section 3.17 which contains references to Eclipse tutorials.

In Section 2.3, we present the Rodin-specific GUI elements.

## 2.3 Installation



**Goals:** The objective of this section is to guide you through downloading, installing and starting Rodin. In addition, we explain the update mechanisms needed to install new plugins for Rodin. Finally, we name the Rodin-specific GUI elements and describe their functions.



**You can skip this section, if...**

- ...you know how to install and update Rodin
- ...you know how to install new plugins for Rodin



Before installing Rodin, you should ensure that your computer fulfills the system requirements. The system requirements for the present release can be found in the Rodin Wiki (3.45).

### 2.3.1 Install Rodin for the first time

#### Step 1: Download

The first step is to download Rodin. Rodin is available for download at the Rodin Download page (3.55)

Rodin is available for Windows, Mac OS, and Linux. No matter which platform you use, the distribution is always packed in a zip-file (3.69). Download the zip file for your system anywhere on your PC.



It is recommended that you download the latest stable version.

## Step 2: Install and Run Rodin

To install Rodin, extract the contents of the zip file in a desired folder. You can run the tool by using the `rodin` executable.

Starting Rodin should bring up the window as shown in figure 2.1. Here you can specify the path where Rodin stores your projects.

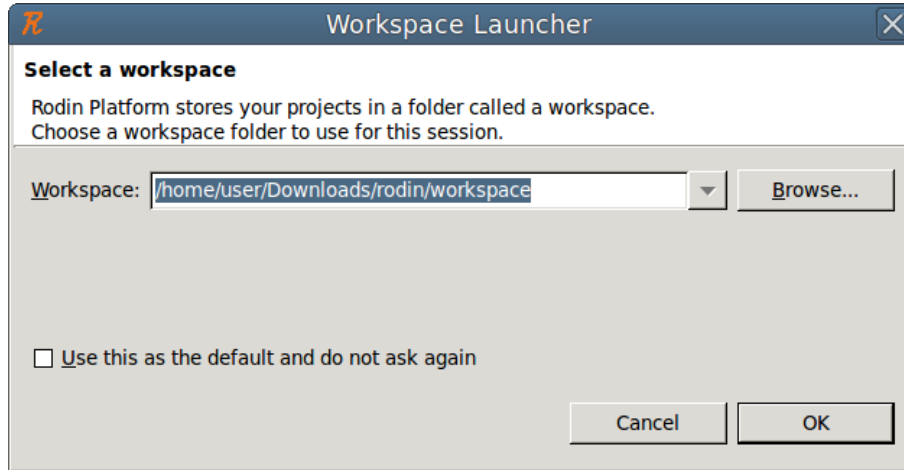


Figure 2.1: Eclipse Workspace Launcher

After specifying a path click on the OK button. Rodin should start and bring up the window shown in figure 2.2.

As already mentioned in Section 2.2.6, the GUI of an Eclipse application consists of Views, Editors, Toolbars, Quickview, Perspectives and many more elements. We name the different Rodin GUI elements (i.e. Views) which are visible after starting Rodin for the first time and explain their functions:

**Menu bar (3.33)** The menu bar of Rodin provides for instance file and edit operations.

**Tool bar (3.64)** The tool bar provides short cuts for familiar commands like save, print, undo and redo.

**Event-B Explorer (3.22)** The Event-B Explorer shows the projects' (3.50) tree structures. It has projects as main entries and for each one its corresponding project files.

**Outline View (3.37)** A view used to view the outline of the active editor or file respectively.

**Rodin Problems View (3.57)** The Rodin Problems View shows problems (i.e. syntax errors) in the active editor.

**Symbols View (3.47)** A view that shows a list of available mathematical symbols which can be used in conjunction with the mathematical notation (3.32).

**Editor View (3.18)** The editor view contains the active editor.

### 2.3.2 Install new plugins

This sections shows how to install new plugins for Rodin by using the example of the Atelier B Provers plugin (3.5). The installation of the provers is highly recommended, because we are not able to prove anything without the provers.

Open the Install Manager of Eclipse `Help > Install New Software...` Select the Atelier B Provers update site in the field `Work with`, mark the Atelier B Provers entry and click on the `Next` button (compare with

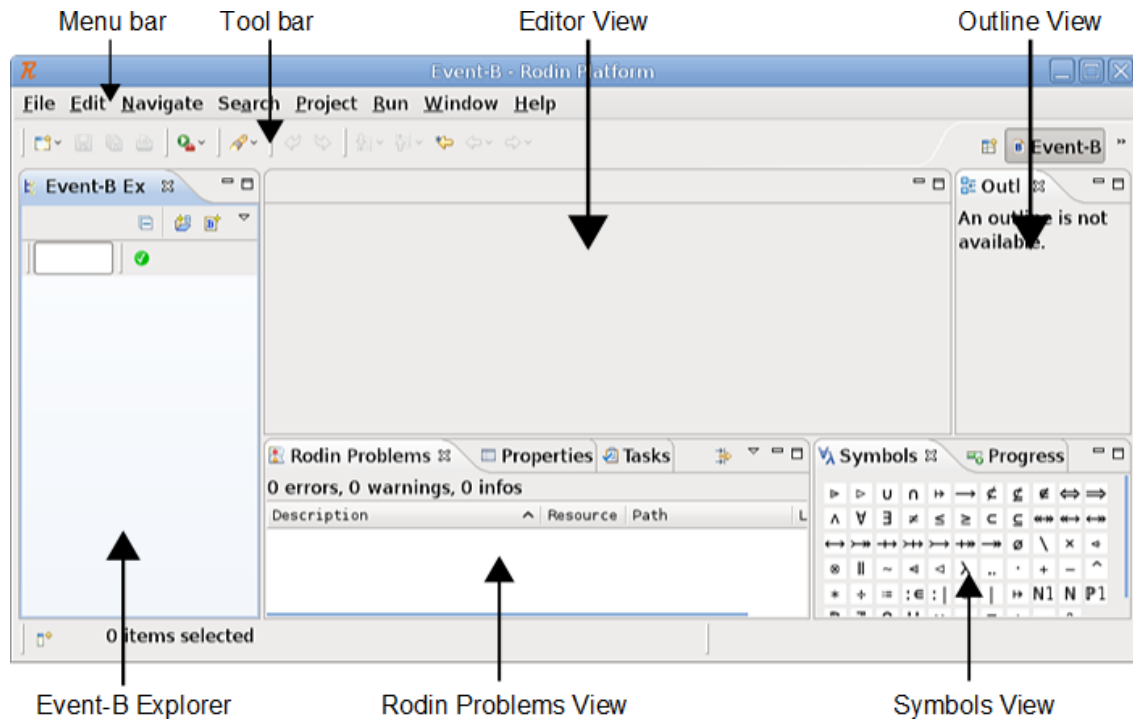


Figure 2.2: Rodin GUI

figure 2.3). Follow the installation instruction to install the plugin. After installing the plugin you will be asked to restart Rodin in order to finalize the installation.

## 2.4 The First Machine: A Traffic Light Controller



**Goals:** The objective of this section is to get acquainted with the modeling environment. We will create a very simple model consisting of just one file to develop a feeling for Rodin and Event-B.

In this tutorial, we will create a model of a traffic light controller. We will use this example in repeatedly in subsequent Sections. Figure 2.4 depicts what we are trying to achieve.

In this Section, we will implement a simplified controller with the following differences:

- We model the signals with booleans to indicated “stop” (false) and “go” (true). We do not model colors (yet).
- We do not model the push button yet.

### 2.4.1 Project Setup

Models typically consist of multiple files that are managed in a Project (3.50). Create a new Event-B Project File > New > Event-B Project. Give the project the name `tutorial-03` as shown in figure 2.5.



Eclipse supports different types of Projects. The Project must have the Rodin Nature (3.58) to work. A project can have more than one nature.

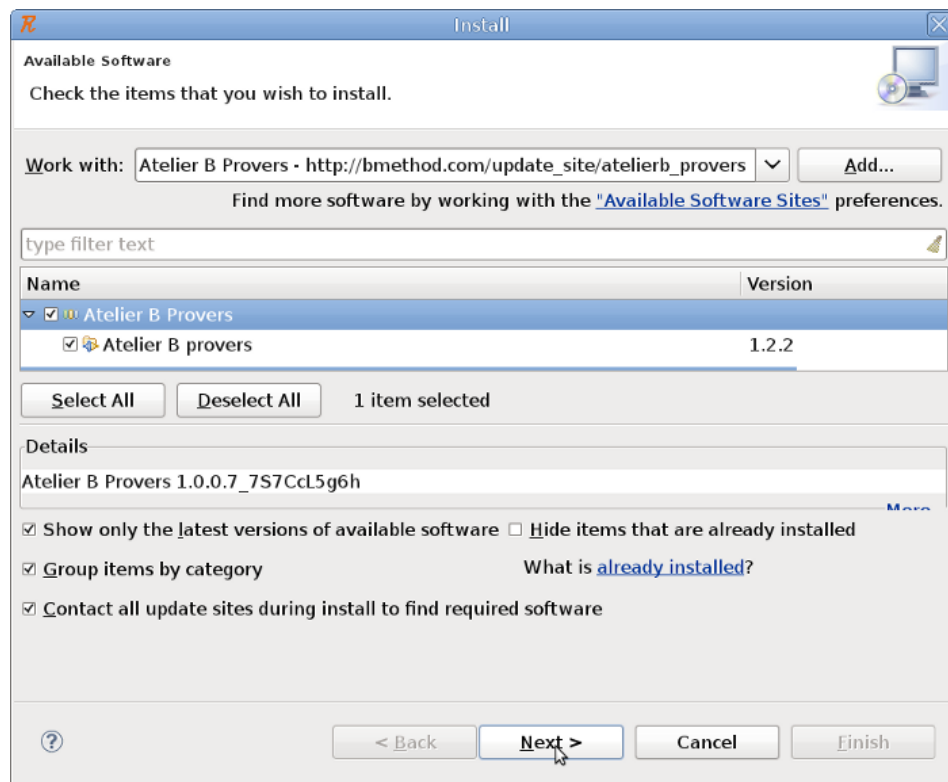


Figure 2.3: Eclipse Install Manager

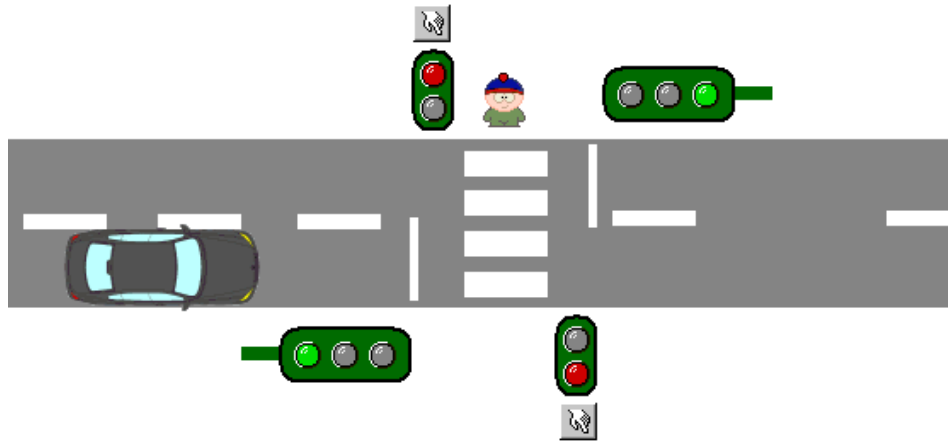


Figure 2.4: The traffic light controller

Next, create a new Event-B Component (3.21). Either use `File > New > Event-B Component`, or right-click on the newly created project and select `New > Event-B Component`. Use `mac` as the component name and click `Finish` as shown in figure 2.6. This will create a Machine (3.31) file.

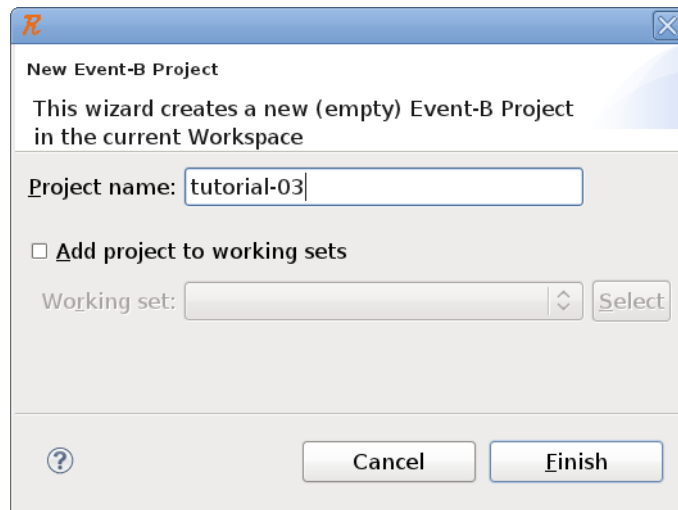


Figure 2.5: New Event-B Project Wizard

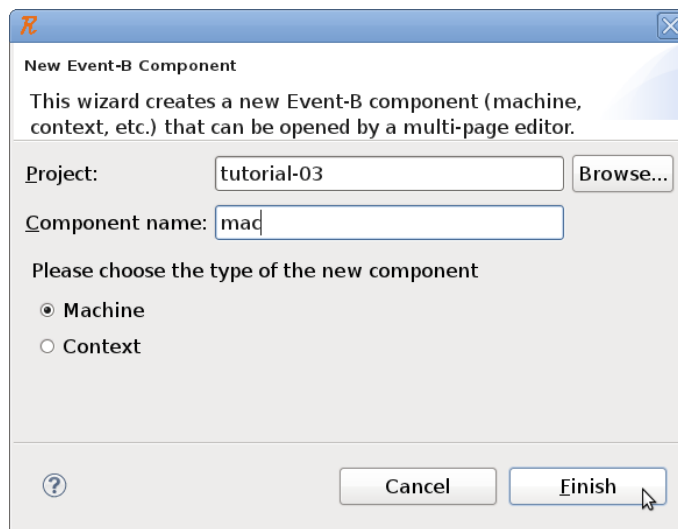


Figure 2.6: New Event-B Component Wizard

The newly created Component will open in the structural editor (3.46). The editor has four tabs at the bottom. The Pretty Print shows the model as a whole with color highlighting, but it cannot be edited here. This is useful to inspect the model. The Edit allows editing of the model. It shows the six main sections of a machine (REFINES, SEES, etc.) in a collapsed state. You can click on the little triangle to the left of a section to expand it.

The editor is *form-based*. This means that in well-defined places an appropriate control (text field, dropdown, etc.) allows modifications.



Alternative editors are available as plug-ins. The form editor has the advantage of guiding the user through the model, but it takes up a lot of real estate and can be slow on big models. The text-based Camille Editor (3.9) is very popular. Please visit the Rodin Wiki (3.45) for the latest

information.

## 2.4.2 Building the Model

Back to the problem: Our objective is to build a simplified traffic light controller, as described in 2.4. We start with the model state. We will model two traffic lights. Therefore, we will create two variables, called `cars_go` and `peds_go`. Go to the **Edit** tab in the editor and expand the **VARIABLES** section. Click on the green plus-sign to create a new variable.

### Creating Variables

You will see two fields, the left one prepopulated with `var1`. Change this to `cars_go`. The second field (after the double-slash “//”) is a comment field that can take any text.



**Comments:** The comment field supports line breaks. Note that it is not possible to “comment out” parts of the model, as you would expect in programming languages. You can use the comment field to “park” predicates and other strings temporarily.

Create the second variable (`peds_go`) correspondingly.

Upon saving, the variables will be highlighted in red, indicating an error as shown in figure 2.7. The Rodin Problems View (3.57) shows corresponding error messages, in this case “Variable `cars_go` does not have a type”.

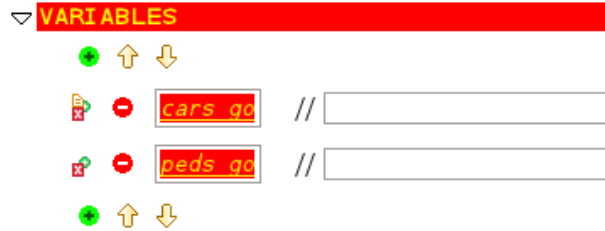


Figure 2.7: Red highlighted elements indicate errors

Types are provided by invariants. Corresponding the steps above, expand the **INVARIANTS** section and add two elements. Invariants have labels (3.30). Default labels are generated (`inv1` and `inv2`). The actual invariant is prepopulated with  $\top$ , representing logical true. Change the first invariant to  $cars\_go \in \text{BOOL}$  and the second invariant to  $peds\_go \in \text{BOOL}$ . Event-B provides the build-in datatype `BOOL`, amongst others (3.14).



**Mathematical Symbols:** Every mathematical symbol has an ASCII-representation that is substituted automatically. To generate “element of” ( $\in$ ), simply type a colon (“:”). The editor will perform the substitution after a short delay. The symbol view shows all supported mathematical symbols. The symbol view discloses the ASCII representation by hovering over the symbol in question.

After saving, you should see that the **EVENTS** section is highlighted in yellow as demonstrated in figure 2.8. Again, the Rodin Problems View gives us the reason: “Variable `cars_go` is not initialized”. Every variable must be initialized in a way that is consistent with the model.

To fix this problem, expand the **EVENTS** and in turn the **INITIALIZATION** event (3.29). Add two elements in the **THEN** block. These are actions that also have labels. In the action fields, provide  $cars\_go := \text{FALSE}$  and  $peds\_go := \text{FALSE}$ .

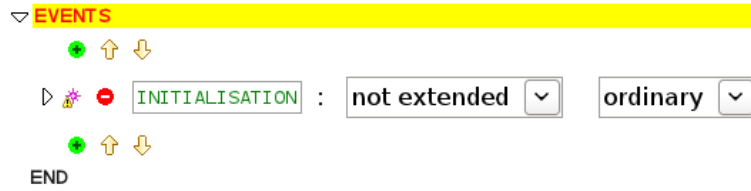


Figure 2.8: Yellow highlighted elements indicate warnings

### State Transitions with Events

Our traffic light controller cannot yet change its state. To make this possible, we provide events (3.19). We start with the traffic light for the pedestrians, and we will provide two events, one to set it to “go” and one to set it to “stop”.



From now on, we won’t describe the individual steps in the editor any more. Instead, we will simply show the resulting model.

The two events will look as follows:



```

Event  set_peds_go  $\hat{=}$ 
  begin
    act1: peds_go := TRUE
  end
Event  set_peds_stop  $\hat{=}$ 
  begin
    act1: peds_go := FALSE
  end

```

### Event parameters

For the traffic light for the cars, we will present a different approach and use only one event with a parameter. The event will use the new traffic light state as the argument. The parameter is declared in the **any** section and typed in the **where** section:



```

Event  set_cars  $\hat{=}$ 
  any
    new_value
  where
    grd1: new_value  $\in$  BOOL
  then
    act1: cars_go := new_value
  end

```

Note how the parameter is used in the action block.

## Invariants

If this model would control a traffic light, we would have a problem, as nothing is preventing the model from setting both traffic lights to **TRUE**. The reason is that so far we only modeled the domain (the traffic lights and their states), and not the requirements. We have the following safety requirement:

REQ-1: Both traffic lights must not be **TRUE** at the same time.

We can model this requirement with the following invariant (please add this invariant to the model):

$$\neg(cars\_go = TRUE \wedge peds\_go = TRUE)$$

Obviously, this invariant can be violated, and Rodin can tell us that. The Explorer (3.22) provides this information in various ways. Go to the explorer and expand the project (**tutorial-03**), the machine (**mac**) and the entry “Proof Obligations”. You should see four proof obligations, two of which are not discharged.



Make sure that you understand the proof obligation labels (3.51). Also, the proof obligations can also be found via other entries. Elements that have non-discharged proof obligations as children are marked with a small question mark. For instance, **inv3** has all proof obligations as children, while the event **set\_cars** has one.

To prevent the invariant from being violated (and therefore to allow all proof obligations to be discharged), we need to strengthen the guards (3.27) of the events.



Before looking at the solution, try to fix the model yourself.

This concludes the tutorial.

### 2.4.3 The Final Traffic Light Model



**MACHINE** mac  
**VARIABLES**

**cars\_go**

**peds\_go**

**INVARIANTS**

**inv1** :  $cars\_go \in \text{BOOL}$

**inv2** :  $peds\_go \in \text{BOOL}$

**inv3** :  $\neg(cars\_go = \text{TRUE} \wedge peds\_go = \text{TRUE})$

**EVENTS**

**Initialisation**

**begin**

**act1** :  $cars\_go := \text{FALSE}$

**act2** :  $peds\_go := \text{FALSE}$

**end**

**Event**  $set\_peds\_go \hat{=}$

**when**

**grd1** :  $cars\_go = \text{FALSE}$

**then**

**act1** :  $peds\_go := \text{TRUE}$

**end**

**Event**  $set\_peds\_stop \hat{=}$

**begin**

**act1** :  $peds\_go := \text{FALSE}$

```

end
Event set_cars  $\hat{=}$ 
  any
    new_value
  where
    grd1 : new_value  $\in$  BOOL
    grd2 : new_value = TRUE  $\Rightarrow$  peds_go = FALSE
  then
    act1 : cars_go := new_value
  end
END

```

## 2.5 Mathematical notation



**Goals:** We want to know how basic properties of a model can be expressed in Event-B. We need a brief introduction to predicates, terms and data types for that.

In Event-B, we use a mathematical notation to describe the systems we want to model. This allows us to be very precise about the model's properties.

### 2.5.1 Predicates

In the traffic light example, we have already encountered several predicates: The invariants of a model and the guards of an event. Also the proof obligations generated by Rodin are predicates.

The simplest predicates are  $\top$  (ASCII: **true**) and  $\perp$  (ASCII: **false**). Also we can check whether arbitrary objects of the same type are equal with  $=$  resp.  $\neq$  (ASCII:  $\neq$ ). Predicates can be combined with the usual logical operators:

|                   | symbol            | ASCII            |
|-------------------|-------------------|------------------|
| conjunction (and) | $\wedge$          | <b>&amp;</b>     |
| disjunction (or)  | $\vee$            | <b>or</b>        |
| implication       | $\Rightarrow$     | <b>=&gt;</b>     |
| equivalence       | $\Leftrightarrow$ | <b>&lt;=&gt;</b> |
| negation (not)    | $\neg$            | <b>not</b>       |

We can use *universal quantification* to express a statement that should hold for all possible values a variable might have. E.g. we want to express that if a number  $x$  is greater than zero, then  $x$  multiplied by 2 is greater than one:

$$\forall x. x > 0 \Rightarrow 2 \cdot x > 1 \quad \text{ASCII: } !x. \ x>0 \Rightarrow 2*x > 1$$

When a variable is introduced by a quantifier, the variables' type must be clear. In this case Rodin can infer that  $x$  must be of type integer, because the operator  $<$  is defined only on integers. Sometimes the type cannot be inferred, e.g. in

$$\forall a, b. a \neq b \Rightarrow b \neq a \quad \text{ASCII: } !a, b. \ a \neq b \Rightarrow b \neq a$$

$a$  and  $b$  could be integers, Booleans or whatever. In that case we must make the type of the variables explicit by stating that  $a$  and  $b$  are elements of the appropriate sets. Let's use integers again:

$$\forall a, b. a \in \mathbb{Z} \wedge b \in \mathbb{Z} \wedge a \neq b \Rightarrow b \neq a \quad \text{ASCII: } !a, b. \ a:\text{INT} \ \& \ b:\text{INT} \ \& \ a \neq b \Rightarrow b \neq a$$

The conjunction operator ( $\wedge$ ) has a stronger binding than the implication  $\Rightarrow$ , so the above is equivalent to

$$\forall a, b. (a \in \mathbb{Z} \wedge b \in \mathbb{Z} \wedge a \neq b) \Rightarrow b \neq a$$



If you are unsure which operators bind stronger, we advise you to use parenthesis to avoid mistakes.

*Existential quantification* on the other hand is used to state that there is an object of a certain type that fulfills a given property. Let's express the statement that there is a Boolean value that is different from TRUE.

$$\exists x. x \in \text{BOOL} \wedge x \neq \text{TRUE} \quad \text{ASCII: } \#x. \ x:\text{BOOL} \ \& \ x \neq \text{TRUE}$$

As you can see, we again added type information for  $x$ . Whereas we put the type information in the universal quantification on the left side of the implication ( $\Rightarrow$ ), for existential quantification we add it via a conjunction ( $\wedge$ ).

### 2.5.2 Data types

We have seen that each identifier (i.e. a variable, constant or parameter) must have a distinguished type. If we can introduce an identifier anywhere, we usually must also add a predicate with which the identifier's type can be determined. In the traffic light example, a variable `cars_go` was introduced and typed by an invariant  $\text{cars\_go} \in \text{BOOL}$ . In the next section we'll see constants that will be typed by axioms (also predicates) and later we'll see parameters that will be typed by guards (again, predicates).

In general, each term in Event-B has a certain type. When saving a Event-B component, Rodin starts the type checker to ensure that types are correctly used. E.g. the terms on both sides of an equality ( $=$ ) must have the same type. If not, Rodin shows an error message. For each type there exists a set that denotes exactly all elements that belong to the type. We now briefly give an overview about all types you might encounter.

**Integers** We have already seen numbers, which are of type integer ( $\mathbb{Z}$ ). Example terms of type  $\mathbb{Z}$  are 5,  $x + 7$  and  $7 \cdot y - 3$ .

**Booleans** We have already seen the Boolean type (BOOL) in the previous section (2.4). It has exactly two elements,  $\text{BOOL} = \{\text{TRUE}, \text{FALSE}\}$ .

**Deferred sets** An user can introduce a new type by adding its name to the **Sets** section of a context. We see that in more detail in the next section (2.6).

**Sets** If we have terms of a certain type, we can easily construct sets of that type. E.g. 1 and  $2 \cdot x$  denote integers ( $\mathbb{Z}$ ) and  $\{1, 2 \cdot x\}$  is a set of integers ( $\mathbb{P}(\mathbb{Z})$ ).  $\mathbb{P}(S)$  (ASCII: POW) denotes the power set (the set of all subsets) of  $S$ .

**Pairs** If we have two terms, we can construct a pair, e.g. with 2 and TRUE, we can construct the pair  $2 \mapsto \text{TRUE}$  (ASCII: 2|->TRUE). The type of that pair is  $\mathbb{Z} \times \text{BOOL}$ , where  $\times$  denotes the Cartesian product.

Set of pairs ("relations") play an important role in modeling languages like Event-B.



Please do not confuse predicates and Boolean values! E.g. if you want to express the condition "if the variable  $b$  is true,  $x$  should be greater than 2", you *cannot* write  $b \Rightarrow x > 2$  (That would raise a syntax error). Instead you can write  $b = \text{TRUE} \Rightarrow x > 2$ .

In the reference section (TODO: link to mathematical notation if the reference) the types of each operator in Event-B are described in detail.

### 2.5.3 Operations on Sets

Let's assume that we have two sets  $A$  and  $B$  of the same type, e.g. sets of integers. Then we can check if an element  $e$  is in it by  $e \in A$  (ASCII: e:A) or on the contrary if it is not in  $A$  is  $e \notin A$  (ASCII: e/:A). Expressing that all elements of  $A$  are also elements of  $B$  (i.e.  $A$  is a subset of  $B$ ) can be done by  $A \subseteq B$  (ASCII: A<:B). The negated form is  $A \not\subseteq B$  (ASCII: A/<:B).

We can build the union  $A \cup B$ , the intersection  $A \cap B$  and the set subtraction  $A \setminus B$ , (ASCII:  $A \setminus B$ ,  $A / \setminus B$  and  $A \setminus B$ ). The set subtraction contains all elements that are in  $A$  but not in  $B$ .

The power set  $\mathbb{P}(A)$  (ASCII:  $\text{POW}(A)$ ) is the set of all subsets of  $A$ . Thus  $B \in \mathbb{P}(A)$  is equivalent to  $B \subseteq A$ .  $\mathbb{P}_1(A)$  (ASCII:  $\text{POW1}(A)$ ) is the set of all non-empty subsets of  $A$ .

### 2.5.4 Introducing user-defined types

We can introduce our own new types simply by giving such types a name. This is done by adding the name of the type to the *SETS* section of a context. How this is done in practice we see in the next tutorial chapter (??).

E.g. if we want to model different kind of fruits in our model, we might add *FRUITS* to our sets. Then the identifier *FRUITS* denotes the set of all elements of this type. Until we do not add any further axioms, nothing more is known about *FRUITS*. Especially, we do not know its cardinality or even if its finite.



Assume that we want to model sub-sets of *FRUITS*, namely *apples* and *oranges*. We do not need not introduce them in the *SETS* section of a context just because they are sets. Let's imagine such a scenario, where *apples* and *oranges* are modeled as types of their own (by declaring them in the *SETS* section). And we have two variables or constants  $a$  and  $o$  with  $a \in \text{apples}$  and  $o \in \text{oranges}$ . Then we cannot compare  $a$  and  $o$  with  $a = o$  or  $a \neq o$ . That would raise a type error because  $=$  and  $\neq$  expect the same type for the left and right expression.

If we want to model sub-sets *apples* and *oranges* like above, we can add them as constants and state  $\text{apples} \subseteq \text{FRUITS}$  and  $\text{oranges} \subseteq \text{FRUITS}$ . If apples and oranges are all fruits we want to model, we can assume  $\text{apples} \cup \text{oranges} = \text{FRUITS}$  and if no fruit is both an apple and orange we can write  $\text{apples} \cap \text{oranges} = \emptyset$ . This can be expressed shorter by saying that apples and oranges constitute a partition of the fruits:  $\text{partition}(\text{FRUITS}, \text{apples}, \text{oranges})$ . In general, we can use the partition operator to express that a set  $S$  is partitioned by the sets  $s_1, \dots, s_n$  with  $\text{partition}(S, s_1, \dots, s_n)$ . We use partitions in Section 2.6.2.

Another typical usage for user defined data types are *enumerated sets*. These are sets where we know already all elements. Let's take a system which can be either working or broken. We want to model that by introducing a type *STATUS* in the *SETS* section and two constants *working* and *broken*. We define that *STATUS* consists of exactly *working* and *broken* by  $\text{STATUS} = \{\text{working}, \text{broken}\}$ . Additionally, we have to say that *working* and *broken* are not the same by  $\text{working} \neq \text{broken}$ .

If the enumerated sets gets larger, we need to state for every two element of the set that they are distinct. Thus, for a set of 10 constants, we'll need  $(10^2 - 10) \div 2 = 45$  predicates. Again, we can use the partition operator to express this more concise:  $\text{partition}(\text{STATUS}, \{\text{working}\}, \{\text{broken}\})$ .

### 2.5.5 Relations

Relations are a powerful instrument when modeling systems. From a mathematical point of view a relation is just a set of pairs. Formally, when we have two sets  $A$  and  $B$ , we can specify that  $r$  is a relation between both by  $r \in \mathbb{P}(A \times B)$  (ASCII:  $r : \text{POW}(A \times B)$ ). Because relations are so common, we can write it shorter  $r \in A \leftrightarrow B$  (ASCII:  $r : A \leftrightarrow B$ ).

With  $a \mapsto b \in r$ , we can check if two elements  $a$  and  $b$  are related in respect to  $b$ .

We use a small example to illustrate relations. Let  $A = \{a, b, c, d\}$  and  $B = \{1, 2, 3, 4\}$ . We define the relation  $r$  with  $r = \{a \mapsto 1, a \mapsto 3, c \mapsto 2, d \mapsto 1\}$ . The *domain* of  $r$  are all elements occurring on the left side  $\text{dom}(r) = \{a, c\}$  and the *range* are all elements on the right  $\text{ran}(r) = \{1, 2, 3\}$ .

To find out to which elements the objects of the set  $s = \{b, c, d\}$  are related to, we can use the *relational image*.  $r[s] = r[\{b, c, d\}] = \{1, 2\}$ . Often we want to know to which object a single element  $b$  is related. We just write it as a singleton set:  $r[\{a\}] = \{1, 3\}$ .

Event-B supports several operators to work with relations. (TODO: link) We don't go more into detail in the course of the tutorial.

An important special case of relations are functions. (TODO: link) Functions are relations where each element of the domain is uniquely related to one element of the range. Event-B directly supports operators to describe partial and total functions, which can be injective, surjective or bijective.

### 2.5.6 Arithmetic

We have the usual operations on integers,  $+$ ,  $-$ ,  $\cdot$  and  $\div$  (ASCII:  $+$ ,  $-$ ,  $*$  and  $/$ ). And they can be compared with the usual  $<$ ,  $\leq$ ,  $\geq$ ,  $>$  (ASCII:  $<$ ,  $\leq$ ,  $\geq$ ,  $>$ ).

$\mathbb{Z}$  (ASCII: INT) denotes the set of all integer numbers,  $\mathbb{N}$  and  $\mathbb{N}_1$  (ASCII: NAT resp. NAT1) are the subsets of natural numbers).



If you specify two variables  $x$  and  $y$  with  $x \in \mathbb{Z}$  and  $y \in \mathbb{N}$ , then both are of type integer ( $\mathbb{Z}$ ).  $\mathbb{N}$  is not another type. There is just the additional condition  $y \geq 0$ .

## 2.6 Introducing Contexts



**Goals:** In this section we introduce Contexts to apply the theoretical concepts that were introduced in the previous section (2.5). We will create a very simple model consisting of just one Context file.

In this tutorial, we will create a model of the well known Agatha puzzle. Here is a brief description of the puzzle:

*Someone in Dreadsbury Mansion killed Aunt Agatha. Agatha, the butler, and Charles live in Dreadsbury Mansion, and are the only ones to live there. A killer always hates, and is no richer than his victim. Charles hates no one that Agatha hates. Agatha hates everybody except the butler. The butler hates everyone not richer than Aunt Agatha. The butler hates everyone whom Agatha hates. No one hates everyone. Who killed Agatha?*

The objective of this Section is to get familiar with Contexts by modeling the Agatha puzzle.

### 2.6.1 Create a Context

Create a new Event-B Project File  $\rangle$  New  $\rangle$  Event-B Project. Give the project the name `tutorial-05`.

Next, create a new Event-B Component (3.21). Unlike in Section (2.4) use `agatha` as the component name and mark the Context (3.12) option in order to create a Context file instead of a Machine (3.31) file.

Click on Finish. Rodin should start the editor with the created Context file (see figure 2.9).

### 2.6.2 Populate the Context

In this section we model the Agatha puzzle step by step.

#### Modelling the Persons

We have three persons in the Agatha puzzle: Agatha itself, the butler and Charles. We model the three persons as constants (one constant for each person) in the corresponding CONSTANTS section:



#### CONSTANTS

Agatha  
butler  
Charles

These constants or persons respectively are part of a set:

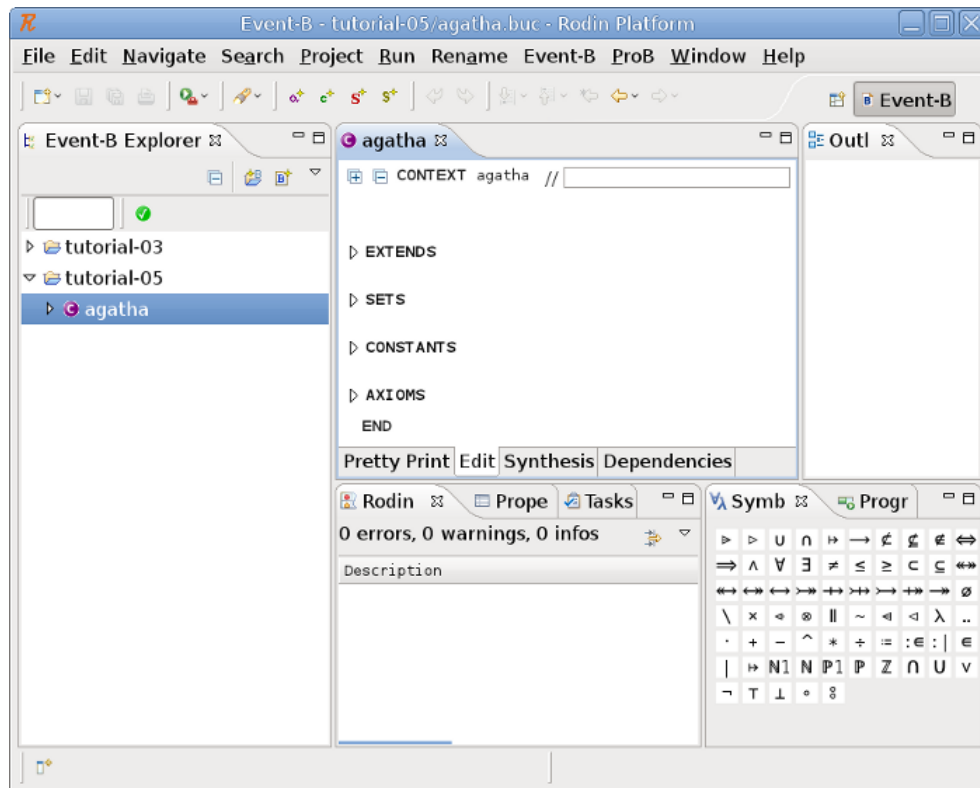


Figure 2.9: Context file opened with Structural Editor

**SETS**

`persons`

Now, the constants itself are not very useful since they have no type (In addition, the constants will be highlighted in red, indicating a problem). The semantics of the sets (3.60) and constants (3.11) are specified in the axioms (3.7). As already mentioned above the persons are part of the set `persons`. We model this by creating a partition (3.39) in the AXIOMS section:

**AXIOMS**

`person_partition : partition(persons, {Agatha}, {butler}, {Charles})`



Please note the curly braces `{}` around the colors. It's very easy to forget these, which will result in typing errors that are very hard to interpret for a novice.



There is a wizard (3.68) for creating the constants, the set and the axiom automatically. Therefore, click on the **New Enumerated Set Wizard** tool bar item. This will bring up the wizard where we can enter the name of the set and the constants in the corresponding text fields. The wizard will create the enumerated set, the constants and the axiom automatically.

### Modelling the Relations “Persons who hate each other” and “Who’s how rich”

We create two more constants `hates` and `richer` to model the relations “Persons who hate each other” and “Who’s how rich”. The relations are abstract, which means that they say nothing about the concrete persons (Agatha, the butler and Charles). We define the concrete relationships between the persons later in this section.

The first constant `hates` is an arbitrary relation (3.53) between `persons`:



#### AXIOMS

`hate_relation`:  $hates \in persons \leftrightarrow persons$

The second constant `richer` is also a relation between `persons`:



#### AXIOMS

`richer_relation1`:  $richer \in persons \leftrightarrow persons$

However, we know that the relation is irreflexive (no person is richer than itself):



#### AXIOMS

`richer_relation2`:  $richer \cap id = \emptyset$

In addition, we know that the relation is transitive:



#### AXIOMS

`richer_relation3`:  $(\forall x, y, z. (x \mapsto y \in richer \wedge y \mapsto z \in richer) \Rightarrow x \mapsto z \in richer)$

Finally, the relation is trichotomous (one person is always richer than the other or vice versa, never both directions):



#### AXIOMS

`richer_relation4`:  $(\forall x, y. x \in persons \wedge y \in persons \wedge x \neq y \Rightarrow (x \mapsto y \in richer \Leftrightarrow y \mapsto x \notin richer))$

### Modelling the “Crime”

Since the objective of the puzzle is to find the killer, we have to create a new constant `killer` which is an element of `persons`:



#### CONSTANTS

`killer`

#### AXIOMS

`killer_type`:  $killer \in persons$

In addition, the puzzle has some more relationships between the different persons which are all modelled as axioms. We know that the killer hates his victim and is no richer than his victim:



#### AXIOMS

`killer_hates`:  $killer \mapsto Agatha \in hates$

`killer_not_richer`:  $killer \mapsto Agatha \notin richer$

Charles hates no one that Agatha hates and Agatha hates everybody except the butler:



#### AXIOMS

```
charles_hates : hates[{ Agatha }] ∩ hates[{ Charles }] = ∅
agatha_hates : hates[{ Agatha }] = persons \ { butler }
```

The butler hates everyone not richer than aunt Agatha and the butler hates everyone whom Agatha hates. However, no one hates everyone:



#### AXIOMS

```
butler_hates_1 : ∀x. (x ↦ Agatha ∉ richer ⇒ butler ↦ x ∈ hates)
butler_hates_2 : hates[{ Agatha }] ⊆ hates[{ butler }] ∧
                (∀x. x ∈ persons ⇒ hates[{ x }] ≠ persons)
noone_hates_everyone : ∀x. x ∈ persons ∧ hates[{ x }] ≠ persons
```

Finally, we have to model the solution:



#### AXIOMS

```
solution : killer = Agatha
```

We mark the axiom `solution` as a theorem (3.63) by clicking on the `not theorem` button as shown in figure 2.10.

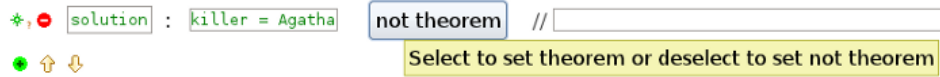


Figure 2.10: Mark an Axiom as Theorem



Theorems describe properties that are expected to be able to be derived from the axioms.

This concludes the tutorial. The following Section shows the complete Context.

### 2.6.3 The Final Context

**CONTEXT** agatha

#### SETS

persons

#### CONSTANTS

Agatha

butler

Charles

hates

richer

killer

#### AXIOMS

```
person_partition : partition(persons, { Agatha }, { butler }, { Charles })
```

```

hate_relation2: hates ∈ persons ↔ persons
richer_relation: richer ∈ persons ↔ persons ∧
    richer ∩ id = ∅ ∧
    (∀x, y, z. (x ↦ y ∈ richer ∧ y ↦ z ∈ richer) ⇒ x ↦ z ∈ richer) ∧
    (∀x, y. x ∈ persons ∧ y ∈ persons ∧ x ≠ y ⇒ (x ↦ y ∈ richer ⇔ y ↦ x ∈
        richer))
killer_type: killer ∈ persons
killer_hates: killer ↦ Agatha ∈ hates
killer_not_richer: killer ↦ Agatha ∉ richer
charles_hates: hates[{Agatha}] ∩ hates[{Charles}] = ∅
agatha_hates: hates[{Agatha}] = persons \ {butler}
butler_hates_1: ∀x. (x ↦ Agatha ∉ richer ⇒ butler ↦ x ∈ hates)
butler_hates_2: hates[{Agatha}] ⊆ hates[{butler}] ∧
    (∀x. x ∈ persons ⇒ hates[{x}] ≠ persons)
noone_hates_everyone: ∀x. x ∈ persons ∧ hates[{x}] ≠ persons
solution: theorem killer = Agatha

```

END

## 2.7 Event-B Concepts

In Event-B we have two kind of components. We already encountered a *context* that describes static elements of a model. On the other side a *machine* describes the dynamic behavior.

### 2.7.1 Contexts

A context has the following components:

**Sets** User-defined types can be declared in the *SETS* section (See also ??).

**Constants** We can declared constants here. The type of each constant must be given by the axioms.

**Axioms** The axioms are a list of predicates. They describe what can be taken for granted when developing a model. The axioms can be later used in proofs that occur in components that use (“see”) this context. Each axiom has a label attached to it.

**Theorems** Axioms can be marked as *theorems*. In that case, we declare that the predicate is provable by using the axioms that are written before the theorem. Theorems can be used later in proofs just like the axioms.

**Extends** A context may extend an arbitrary number of other contexts. Extending another context *A* has the effect that we can make use of all constants and axioms declared in *A* and we can add new constants and axioms.

Rodin automatically generates *proof obligations* (often abbreviated as PO) for properties that should be proven. Each proof obligation has a name that explains where it comes from. There are two kind of proof obligations generated in a context:

- Each theorem must be proven. The proof obligation’s name has the form *label/THM*, where *label* is the theorem’s label.
- Some expressions are not obviously *well-defined*. E.g. the axiom  $x \div y > 2$  is only meaningful if *y* is different from 0. Thus Rodin generates the proof obligation  $y \neq 0$ . A well-definedness proof obligation has the name *label/WD*.

Because the proof of a theorem or well-definedness may depend on the axioms and theorems written before, their order matters. This is necessary to avoid circular reasoning.

### 2.7.2 Machines

A machine describes the dynamic behavior of a model by means of variables whose values are changed by events. A central aspect of modeling a machine is to prove that the machine never reaches an invalid state, i.e. the variables always have values that satisfy the invariant. First we briefly summarize of which parts a machine consists:

**Refines** Optionally a machine can refine another one. We'll see in 2.7.4 what that means.

**Sees** We can use the context's sets, constants and axioms in a machine by declaring it in the *Sees* section. The axioms can be used in every proof in the machine as hypothesis.

**Variables** The variables' values are determined by an initialization and can be changed by events. Together they constitute the state of the machine. Each variable must be given a type by the invariants.

**Invariants** These are predicates that should be true for every reachable state. Each invariant has a label.

**Events** An event can assign new values to variables. The *guards* of an event specify under which conditions it might occur. The initialization of the machine is a special case of an event.

### 2.7.3 Events

We saw in 2.4 how an event basically looks like using the example of a traffic light:

```
Event  set_cars  $\hat{=}$ 
    any
        new_value
    where
        grd1 : new_value  $\in$  BOOL
    then
        act1 : cars_go := new_value
    end
```

We have the event's name *set\_cars*, a *parameter* *new\_value*, a *guard* with label *grd1* and an *action* with label *act1*. An event can have an arbitrary number of parameters, guards and events.

The guards specify *when* an event might occur, i.e. under which combinations of the values of the machine's variables and the parameters. The actions describe *what* changes apply to the variables.

Only the variables that are explicitly mentioned in the actions are affected. All the other variables keep their old values. Beside the simple assignment ( $:=$ ), there are other forms of actions ( $:\in$  or  $:|$ ) which are explained in the reference section ??.

The *initialization* of the machine is a special form of event. It has neither parameters nor guards.

Now our aim is to prove that the invariants always hold. To do this, we must prove two things:

- The initialization leads to a state where the invariant holds.
- Assuming that the machine is in a state where the invariant holds, every enabled event leads to a state where the invariant holds.

Rodin generates proof obligations for every invariant that is possibly affected by an event. I.e. that the invariant uses variables that are changed by the event. The name of the proof obligation is then *event\_name/invariant\_label/INV*.

The goal of such a proof is the affected invariant where all variables are replaced by the new values from the actions. We can use the following hypothesis:

- the unchanged invariants because we now that before the event the invariants were true
- the guards because the event only occurs when the guards are true

In the special case of an initialization we cannot use the invariants, because we do not make any assumptions about uninitialized machines.

#### 2.7.4 Refinement

Refinement is a central concept in Event-B. Refinements are used to gradually introduce the details and complexity into a model. If a machine **B** refines a machine **A**, **B** is only allowed to behave in a way that corresponds to the behavior of **A**. We now look into more detail what “corresponds” here means. In such a setting, we call **A** the abstract and **B** the concrete machine.

We give here just a brief overview over the concept of refinement, in the later section 2.8 we use refinement in an example.

The concrete machine has its own set of variables. Its invariants can refer to the variables of the concrete and the abstract machine. If a invariant refers to both, we call it a “gluing invariant”. The gluing invariants are used to relate the states between the concrete and abstract machines.

The events of a concrete machine can now refine an abstract event. To provide that the concrete machine does only what the abstract one allows, we must show two things:

- The concrete events can only occur when the abstract one can occur.
- If a concrete event occurs, the abstract event can occur in such a way that the resulting states correspond again, i.e. that the gluing invariant remains true.

The first condition is called “guard strengthening”. The resulting proof obligation has the label `concrete_event/abstract_guard/GRD`. We have to prove that under the assumption that the concrete event is enabled (i.e. its guard are true) and the invariants (both the abstract and the concrete) hold, the abstract guards holds, too. Thus the goal of the proof is the abstract guard and the invariants and concrete guards can be used as hypothesis in the proof.

The second condition that the gluing invariant remains true is just a more general case of the proof obligation that an event does not violate the invariant. So the proof obligation’s label is again `concrete_event/concrete_invariant/INV`. The goal of the proof is the invariant of the concrete machine where each occurrence of a modified variable is replaced by its new value. The hypothesis we can use are:

- The invariant of both the concrete and abstract machines: We assume that they hold before the event occurred.
- The abstract invariants where the modified variables are replaced by their new values, because we know that the abstract event does not violate the invariants, as we have proven before.
- The guards of both the concrete and abstract machines: The event occurs only when they are true.

These two conditions are the central piece to prove the correctness of a refinement. We now just explain a few common special cases.

##### Variable re-use

Most often, we do not want replace all variables by new ones. It is sometimes useful to keep even all variables. We can do this just by repeating the names of the abstract variables in the variable section of the concrete machine. In that case we must prove for each concrete event that changes such a variable that the corresponding abstract event updates the variable in the same way. The proof obligation has the name `concrete_event/abstract_action/SIM`.

### Introducing new events

An event in the concrete machine might refine no event in the abstract machine. In that case it is assumed to refine *skip*, the event that does nothing and can occur any time. The guard strengthening is then trivially true and doesn't need to be proven. It still must be proven that the gluing invariant holds, this time under the assumption that the abstract machine's variables have not changed. So the new state of our newly introduced event corresponds to the same state of our abstract machine than before the event happened.

### Witnesses

Let's assume a situation where we have an abstract event with a parameter  $p$  and in a refining event we do not need that parameter anymore. Above we saw that we have to prove that for each concrete event the abstract event may act accordingly. But with the parameter, we now have the situation that we must prove the existence of a value for  $p$  such that an abstract event exists. Proofs with existential quantification are often hard to do so Event-b use the construct of a *witness*. A witness is just a predicate on the abstract parameter with the name of the variable as label. Often a witness has just the simple form  $p = \dots$ .

## 2.8 Expanding the Traffic Light System: Contexts and Refinement

✓ **Goals:** We apply what we learned in the previous section by introducing a context with traffic light colors, and a refinement to integrate them. We will introduce another refinement for the push buttons.

### 2.8.1 Data Refinement

We will continue the example from Section 2.4, where we built a simplified model of a traffic light controller. The model was simplified because we abstracted the traffic lights to TRUE and FALSE, and a number of features were still missing.

We will introduce data refinement in this section. The objective is to create a mapping between the abstract traffic light values and actual colors. Figure 2.11 depicts our mapping for the traffic light.

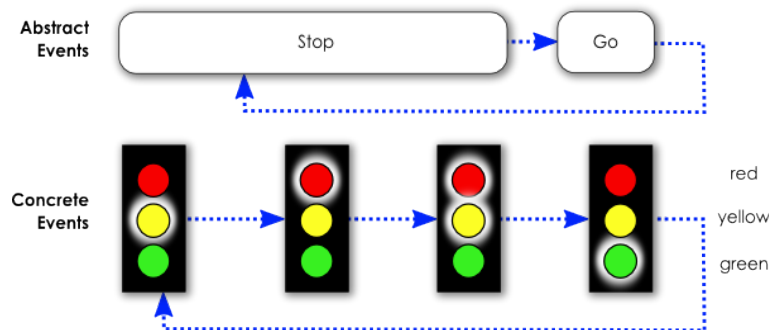


Figure 2.11: Mapping between Abstract and Concrete Events

For simplicity, the traffic light for pedestrians consists of only two lights, red and green. We break this problem into two steps:

1. Create a context with the data structures for the colors
2. Create a refinement of the existing model that sees the new context and refines the boolean states into colors.

### 2.8.2 A Context with Colors

Start by creating a context called `ctx1`, as described in Section 2.6. We model the colors as constants:



#### CONSTANTS

```
red
yellow
green
```

These constants are part of a set:



#### SETS

```
COLORS
```

And last, we need to provide typing. We do this by creating a partition (3.39):



#### AXIOMS

```
type : partition(COLORS, {red}, {yellow}, {green})
```



Please note the curly braces `{}` around the colors. It's very easy to forget these, which will result in typing errors that are very hard to interpret for a novice.

This completes the context.

### 2.8.3 The Actual Data Refinement

The easiest way to create a refinement is by right-clicking on the machine in the project browser and to select Refine. This will create a “stub” consisting of all variables and events.

TODO: Screenshot of selecting the context menu.



In refinements, the Edit View of the Editor hides some information from the abstract machine by default. This can be particularly important when you modify a refined event. The Pretty Print View shows all element, with elements from the abstract machine in *italics*.

First we have to make the machine aware of the context by adding a `sees` (3.59) statement:



```
MACHINE mac1
REFINES mac0
SEES ctx1
```

We start with the traffic lights for the pedestrians, which consists of just two colors (red and green). We introduce a new variable to represent the traffic light colors called `peds_colors`, with corresponding typing and an initialization (only the changes are shown in the following):



#### VARIABLES

```
peds_colors
```

#### INVARIANTS

```
inv4 : peds_colors ∈ {red, green}
```

#### EVENTS

```

Initialisation
  extended
  begin
    init4: peds_colors := red
  end
END

```

Next, we will create a gluing invariant (3.25) that associated `peds_go` from the abstract machine with the variable `peds_colors` that we just created. The gluing invariant shall map `TRUE` to `green` and `FALSE` to `red`:



### INVARIANTS

```
gluing: peds_go = TRUE ⇔ peds_colors = green
```

In its current state, this gluing invariant can be violated: if the event `set_peds_go` is triggered, for instance, the variable `peds_go` will change, but not `peds_colors`. We expect that this will result in undischarged proof obligations 3.65. We can check this by expanding the machine in the Event-B Explorer. Indeed, we now see two undischarged proof obligations (compare with figure 2.12).

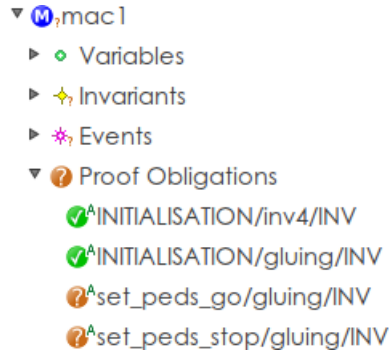


Figure 2.12: Mapping between Abstract and Concrete Events

To fix this, we have to modify the two events in question. Let's start with `set_peds_go`. First, we change the event from extended to not extended in the Editor as shown in figure 2.13.

This change will copy the guard and action from the abstract machine, so that we can modify it. We can now replace the action with the corresponding action regarding `peds_colors` (setting it to `go`). While we are at it, we can also rename the name of the event to something more fitting (e.g. `set_peds_green`).

Next, we perform the corresponding change on `set_peds_stop`. And last, `set_cars` also contains a reference to `peds_go` that must be replaced.

Once all references to `peds_go` have been replaced, we can remove the variable `peds_go` from the `VARIABLES` section. You shouldn't have any errors or warnings, and all proof obligations should be discharged.



If you get the error message "Identifier `peds_go` has not been declared", then there are references to the refined variable left somewhere in the model. It can be helpful to use the `Pretty Print` view, as it will show the "inherited" elements from the abstract machine as well.

#### 2.8.4 The refined machine with data refinement for `peds_go`



```

MACHINE mac1
REFINES mac0

```

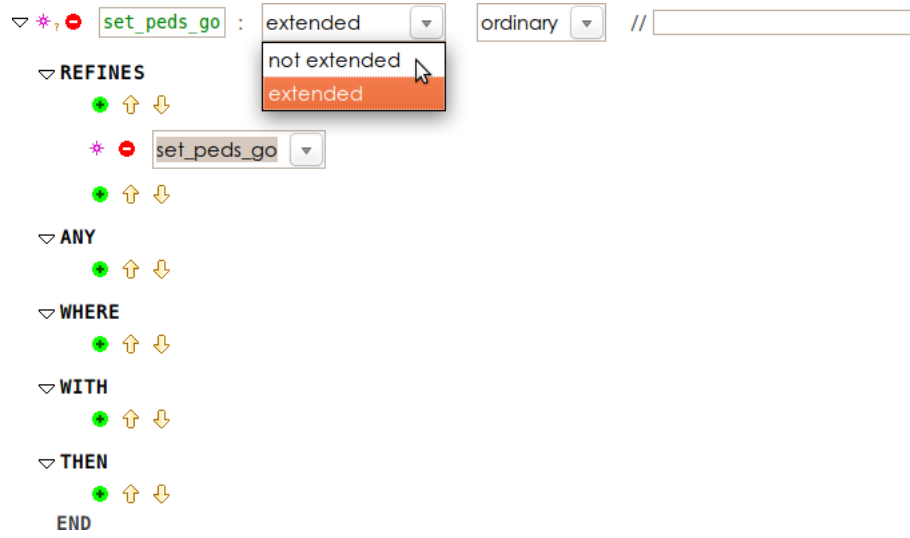


Figure 2.13: Switch from extended to not extended

```

SEES ctx1
VARIABLES
  cars_go
  peds_colors
INVARIANTS
  inv4 : peds_colors ∈ {red, green}
  gluing : peds_go = TRUE ⇔ peds_colors = green
EVENTS
Initialisation
  begin
    act1 : cars_go := FALSE
    init4 : peds_colors := red
  end
Event set_peds_green ≐
refines set_peds_go
  when
    grd1 : cars_go = FALSE
  then
    act2 : peds_colors := green
  end
Event set_peds_red ≐
refines set_peds_stop
  begin
    act1 : peds_colors := red
  end
Event set_cars ≐
refines set_cars
  any
    new_value
  where
    grd1 : new_value ∈ BOOL

```

```

    grd2 : new_value = TRUE  $\Rightarrow$  peds_colors = red
  then
    act1 : cars_go := new_value
  end
END

```

### 2.8.5 Witnesses

The refinement of `set_cars` is more difficult, as the event uses a parameter (the new value for `cars_go`). In order to create the association, we need a witness (3.67).

A witness is for an event's parameter what a gluing invariant is for a variable: it is a mapping between the abstract parameter and the new parameter and allows the abstract parameter to disappear. In this example, the abstract parameter `new_value` is of type `BOOL`, while we introduce a new parameter `new_value_colors` of type `COLORS`.



The naming of a witnesses' label is mandatory and must be the name of the abstract parameter. In our example, the label must be `new_value`

So let's get started. We first provide the new variable, gluing invariant, typing invariant and initialization as before. Note that the traffic light for the cars can show more than one color at a time. Therefore, the variable contains a set of colors, instead of just one color (as modelled for `PEDS_COLORS`):



#### VARIABLES

```
cars_colors
```

#### INVARIANTS

```
inv5 : cars_colors  $\subseteq$  COLORS
```

```
gluing_cars : cars_go = TRUE  $\Leftrightarrow$  green  $\in$  cars_colors
```

#### EVENTS

##### Initialisation

```
begin
```

```
  init5 : cars_colors := {red}
```

```
end
```

```
END
```

We also have to modify the guard on `set_peds_green`, something that you should now be able to figure out yourself.

The interesting piece is the last event, `set_cars`, which we rename into `set_cars.colors`. We change the parameter to `new_value.colors` and type it as a subset of `COLORS`.

The witness appears in the `with` section of the event. The label **must** be `new_value`. The value itself must describe the relationship between the abstract parameter `new_value` and the new parameter `new_value.colors`. As we use the parameter as the new value for the variable `cars_colors`, the witness is an adaptation of the gluing invariant (we just replace `cars_colors` with `new_value.colors`).



In most cases, the witness is a slightly modified gluing invariant.

Here is the resulting event:



```

Event  set_cars.colors  $\hat{=}$ 
refines set_cars
any
  new_value.colors

```

```

where
  grd1 : new_value_colors ⊆ COLORS
  grd2 : green ∈ new_value_colors ⇒ peds_colors = red
with
  new_value : new_value = TRUE ⇔ green ∈ new_value_colors
then
  act1 : cars_colors := new_value_colors
end

```

Even though all proof obligations are now discharged, we're not done yet: While the traffic light doesn't violate the safety property from the abstract machine, it doesn't behave the way described in Section 3.13. We still have to ensure that the lights are activated in the proper sequence. We can impose this behavior by adding four guards, with each guard defining one transition:



```

grd_y_r : cars_colors = {yellow} ⇒ new_value_colors = {red}
grd_r_ry : cars_colors = {red} ⇒ new_value_colors = {red, yellow}
grd_ry_g : cars_colors = {red, yellow} ⇒ new_value_colors = {green}
grd_g_y : cars_colors = {green} ⇒ new_value_colors = {yellow}

```

### 2.8.6 Discussion

Note that we used two very different approaches to model the traffic lights for cars and pedestrians. For the pedestrians, we created one event for each state transition. For the cars, we handled all states in one single event.

You will often be confronted with situations where many modelling approaches are possible. You should consider two main factors when modelling: (1) readability of the model and (2) ease of proof. In this case, both approaches are equally good (although we wouldn't recommend mixing different approaches in one model. We did it here only to demonstrate both approaches).

We won't cover deadlocks (3.15) and deadlock freedom in this tutorial; however, we want to point out that the traffic light for the cars is interesting regarding deadlocks. Consider `cars_colors = { green, red }`. This is a legal state, but it would block `set_cars_colors` forever! A model checker (like ProB (3.49)) could find it. This is not a problem, however, as with the given initialization and events, that state is not reachable in the first place.

We hope that this Section helped you to understand the power of abstraction. The safety invariant from Section 2.4.2 was very simple. We could now introduce colors, being confident that the invariant will still hold (assuming, of course, that our gluing invariant is correct).

### 2.8.7 The Refined Machine with All Data Refinement



```

MACHINE mac1
REFINES mac0
SEES ctx1
VARIABLES
  peds_colors
  cars_colors
INVARIANTS
  inv4 : peds_colors ∈ {red, green}
  inv5 : cars_colors ⊆ COLORS
  gluing_peds : peds_go = TRUE ⇔ peds_colors = green
  gluing_cars : cars_go = TRUE ⇔ green ∈ cars_colors
EVENTS

```

```

Initialisation
  begin
    init4: peds_colors := red
    init5: cars_colors := {red}
  end
Event set_peds_green  $\hat{=}$ 
refines set_peds_go
  when
    grd1: green  $\notin$  cars_colors
  then
    act2: peds_colors := green
  end
Event set_peds_red  $\hat{=}$ 
refines set_peds_stop
  begin
    act1: peds_colors := red
  end
Event set_cars_colors  $\hat{=}$ 
refines set_cars
  any
    new_value_colors
  where
    grd1: new_value_colors  $\subseteq$  COLORS
    grd2: green  $\in$  new_value_colors  $\Rightarrow$  peds_colors = red
    grd_y_r: cars_colors = {yellow}  $\Rightarrow$  new_value_colors = {red}
    grd_r_ry: cars_colors = {red}  $\Rightarrow$  new_value_colors = {red, yellow}
    grd_ry_g: cars_colors = {red, yellow}  $\Rightarrow$  new_value_colors = {green}
    grd_g_y: cars_colors = {green}  $\Rightarrow$  new_value_colors = {yellow}
  with
    new_value: new_value = TRUE  $\Leftrightarrow$  green  $\in$  new_value.colors
  then
    act1: cars_colors := new_value.colors
  end
END

```

### 2.8.8 One more Refinement: The Push Button

We will demonstrate another application of refinement: Introducing new features into an existing model. A typical traffic light system allows the pedestrians to request crossing the street by pressing a button. We will introduce this feature in a new refinement.

While we could have introduced the push button in the initial machine, introducing it later allows us to structure the model, making it easier to understand and navigate.

We will realize this feature by introducing a new boolean variable for the push button. We will introduce an event that notifies the model of the pressing of the push button. Upon allowing the pedestrians to cross, we will reset the push button.

As in the previous Section, we create a new refinement *mac2* by right-clicking on *mac1* and selecting Refine. A stub is generated that contains the events from the abstract machine. We can simply add a new variable for the push button (including typing and initialization). We also introduce an event to set the button. This event doesn't work while the pedestrians have a green light.



#### VARIABLES

*button*

```

INVARIANTS
  type_button: button ∈ BOOL
EVENTS
Initialisation
  extended
  begin
    init_button: button := FALSE
  end
Event push_button ≡
  when
    grd: peds_colors = red
  then
    act: button := TRUE
  end
END

```

What's left now is the integration of the push button with the traffic light logic. This includes:

- Upon pressing the button, the pedestrians must eventually get a green light
- At some point, the button variable must be reset

As we will see in the following discussion, this can get more tricky than it first appears. For now, we will introduce a guard preventing the car lights to turn to green when button is true. And we will reset the button when the pedestrian lights turn red:



```

Event set_peds_red ≡
extends set_peds_red
  begin
    act_button: button := FALSE
  end
Event set_cars_colors ≡
extends set_cars_colors
  where
    grd_button: ¬(cars_colors = {red} ∧ button = TRUE)
  end

```

### 2.8.9 Discussion

There are a number of problems associated with the model in its current state. Let's start with the resetting of the button: The way we built our model so far, `set_peds_red` can be triggered any time - there is not a single guard, neither in the abstract event. Therefore, the button could be reset any time, without the pedestrian light ever turning green.

This could be prevented with additional guards - for instance, the traffic light events could require an actual change in the light's status. This in turn could lead to deadlocks.

But even if we introduce such guards, we could get stuck in a situation where cars would never get green light any more. Consider the following scenario: (1) pedestrians get green light; (2) the light turns red; (3) a pedestrian presses the button again; (4) this prevents the car lights from turning green, instead the pedestrians get green light again and the cycle can start over.

There are tactics to address all these issues. However, rarely is it possible to generate proof obligations for such scenarios (without making the model much more complicated). It can be useful to use model checkers 3.34 to validate the model's behaviour, or even to use temporal logic 3.62 to articulate properties of the model.



As an exercise, try to improve the model to address these issues.

## 2.9 Proving

✓ **Goals:** The goal of this section is to get familiar with the Proving Perspective and to prove a simple proof by hand.

### 2.9.1 The Celebrity Problem

In this section we will work on the model of the so-called celebrity problem. In the setting for this problem, we have a “knows” relation. This relation is defined so that

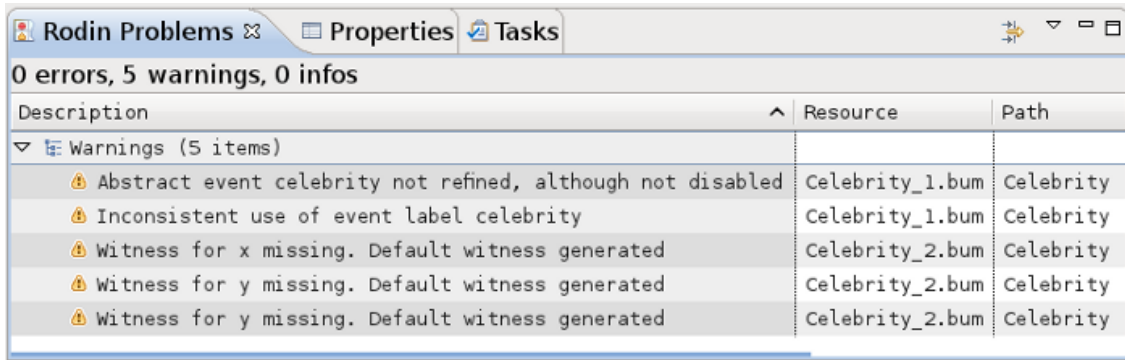
- no one knows himself,
- the celebrity knows nobody, but
- everybody knows the celebrity.

The goal is to find the celebrity.

⚠ Make sure that you have no existing Project named Celebrity. If you do, then rename it. For this right click on the project and select **Rename...**

Import the archive file **Celebrity.zip** to you Event-B Explorer. For this, choose **File > Import > General > Existing Project into workspace**. Then, select the according archive file and click on **Finish**.

Rodin takes a few seconds to extract and load all the files. Once it is done, it shows that there are a few problems with this project as you can see in the Rodin Problems View (compare with figure 2.14).



| Rodin Problems 53 Properties Tasks                            |                 |           |
|---------------------------------------------------------------|-----------------|-----------|
| 0 errors, 5 warnings, 0 infos                                 |                 |           |
| Description                                                   | Resource        | Path      |
| ▼ Warnings (5 items)                                          |                 |           |
| ⚠ Abstract event celebrity not refined, although not disabled | Celebrity_1.bum | Celebrity |
| ⚠ Inconsistent use of event label celebrity                   | Celebrity_1.bum | Celebrity |
| ⚠ Witness for x missing. Default witness generated            | Celebrity_2.bum | Celebrity |
| ⚠ Witness for y missing. Default witness generated            | Celebrity_2.bum | Celebrity |
| ⚠ Witness for y missing. Default witness generated            | Celebrity_2.bum | Celebrity |

Figure 2.14: Warnings in the Rodin Problems View

In the first part of this section, our goal is to fix these problems. Let’s take a look at the warning stating that the event label “celebrity” is misused (“Inconsistent use of event label celebrity”). Double-click on the warning to open the **Celebrity\_1** machine. The problem is that the event is not declared as a refinement. To solve the problem, expand the event and add a new entry in the **REFINES** section (Click on the green plus-sign to create a new entry). Select the event **celebrity**. This declares that the event is a refinement of an event with the same name in the abstract machine (3.2). As this is the case here, we can now save the project and the warning should disappear.

The three remaining warnings state that witnesses (3.67) are missing. So, double click on the warning to open the concrete model (here **Celebrity\_2**). Then, expand the **celebrity** event and add a witness in the **WITH** section (Click on the green plus-sign to create a new entry).

A default witness **wit1** has been created, with a default value  $\top$  (e.g. the predicate “true”) which we need to change. Its name will have to be **x** if we want it to be a witness for the parameter **x** of the corresponding abstract event in the machine **Celebrity\_1**. The abstract event has the assignment  $r := x$ , while the concrete one has the assignment  $r := b$ . So the content of the witness is  $x = b$ . The event should now look as follows:

```

Event  celebrity  $\hat{=}$ 
refines celebrity
  when
    grd1 :  $R = \emptyset$ 
  with
    x :  $x = b$ 
  then
    act1 :  $r := b$ 
  end

```

Edit the content and save the file. One warning will disappear, two to go.



Try completing the other two witnesses on your own. A hint: Both witnesses are simple equalities, and both can be found by comparing the third guard of the abstract event with the second guard of the concrete one. Remember to give the witness the name of the variable it stands for. If you completed this step correctly, there should be no warning, info or error left on the Rodin Problems View (3.57).

The following section 2.9.2 shows the final `Celebrity_2` machine.

## 2.9.2 The Final Second Refinement

```

MACHINE  Celebrity_2
REFINES  Celebrity_1
SEES    Celebrity_c0
VARIABLES
  r
  R
  b
INVARIANTS
  inv1 :  $R \subseteq P$ 
  inv2 :  $b \in P$ 
  inv3 :  $b \notin R$ 
  inv4 :  $Q = R \cup \{b\}$ 
EVENTS
Initialisation
  begin
    act1 :  $r \in P$ 
    act2 :  $b, R : |b' \in P \wedge R' = P \setminus \{b'\}$ 
  end
Event  celebrity  $\hat{=}$ 
refines celebrity
  when
    grd1 :  $R = \emptyset$ 
  with
    x :  $b = x$ 
  then
    act1 :  $r := b$ 
  end

```

```

Event  remove_1  $\hat{=}$ 
refines remove_1
  any
     $x$ 
  where
    grd1 :  $x \in R$ 
    grd2 :  $x \mapsto b \in k$ 
  with
     $y : b = y$ 
  then
    act1 :  $R := R \setminus \{x\}$ 
  end
Event  remove_2  $\hat{=}$ 
refines remove_2
  any
     $x$ 
  where
    grd1 :  $x \in R$ 
    grd2 :  $x \mapsto b \notin k$ 
  with
     $y : b = y$ 
  then
    act2 :  $b := x$ 
    act1 :  $R := R \setminus \{x\}$ 
  end
END

```

### 2.9.3 The First Proof

In this section we prove the model of the Celebrity Problem. Therefore, switch to the Proving Perspective. You can switch between perspectives using the shortcut bar as shown in figure 2.15.

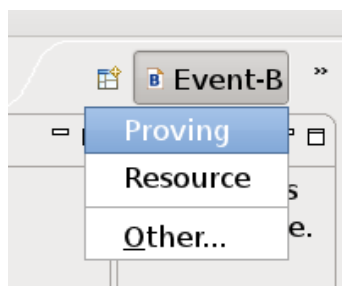


Figure 2.15: Switch Perspective



If the Proving Perspective is not available in the menu, select Other... ) Proving. This will open a new window which shows all available Perspectives.

We should now see the window in figure 2.16.

The **Proving Perspective** contains three new important views:

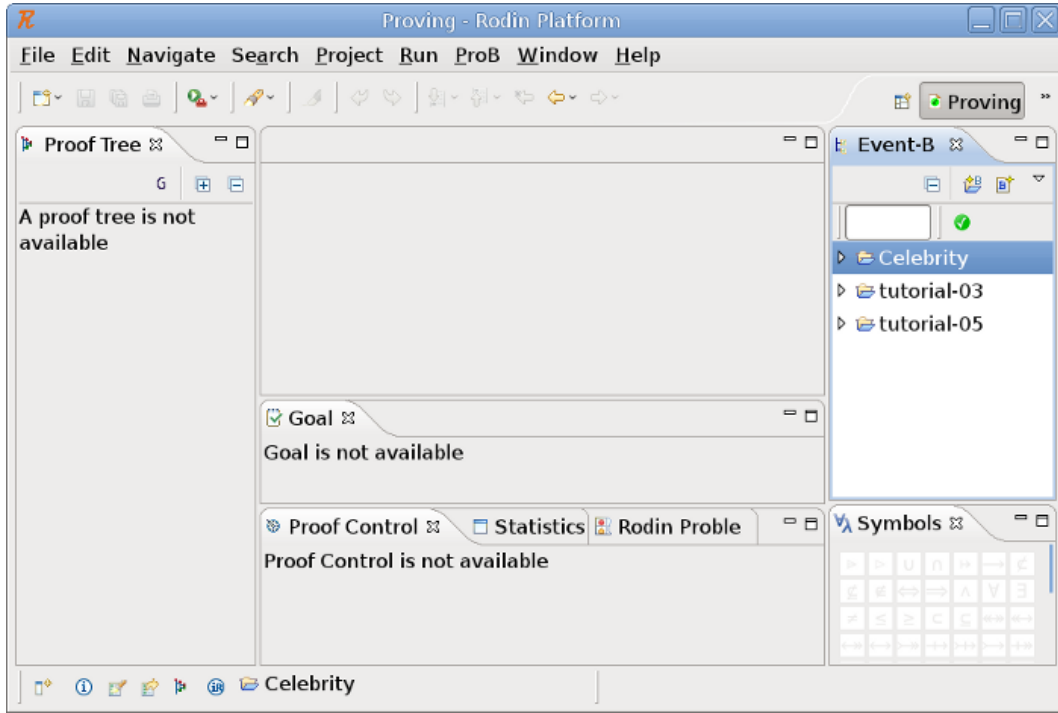


Figure 2.16: Rodin Proving Perspective

**Proof Tree View (3.42)** Here we see a tree of the proof that we have done so far and the current position in it. By clicking in the tree, we can navigate inside the proof. Currently, we have not started with the proof yet, so there is no new place to move to.

**Proof Control View (3.40)** This is where we perform interactive proofs.

**Goal View (3.26)** This window shows what needs to be proved at the current position inside the proof tree.

Expand the `Celebrity_1` machine in the `Event-B Explorer`. After this expand the `Proof Obligations` section. We can see that the auto prover (3.6) did quite a good job. All except for three proofs already should be completed (indicated by a green chop).

Let's start with the proof `remove_1/inv2/INV` of `Celebrity_1`. For this double click on the proof obligation `remove_1/inv2/INV`.

We should now see the window in figure 2.17.



Make sure that you understand the different buttons in the **Proof Control View (3.40)**.

In order to prove the proof obligation, it suffices to prove  $x \neq c$ . So type this into the **Proof Control View (3.40)** and press the **Add Hypothesis** button (ah).



In order to revert a step click on a node in the **Proof Tree View** and click on the scissors button in the **Proof Control View** or open the context menu of a node and select **Prune**.

The new goal is  $\neg x = c$ . Now, try selecting the right hypotheses by yourself in order to complete the proof. For this click on the **Search hypothesis** button in the **Proof Control View**. On the left side we should

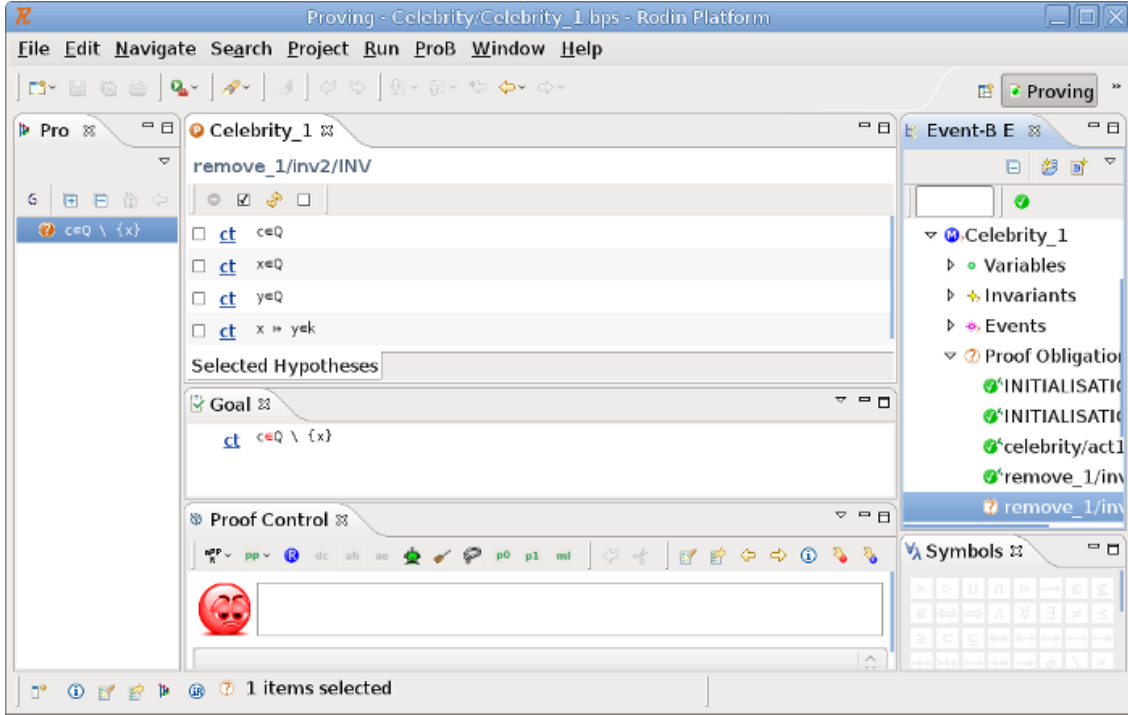


Figure 2.17: Proof Obligation

see now the **Selected Hypotheses View** (compare with figure 2.18). If you cannot find the right hypotheses, you may also just select all hypotheses. To add the selected hypothesis to the **Selected Hypotheses View** just click on the green plus button.

Click now on the **p0** button to prove the goal  $\neg x = c$  with the **Predicate prover** on selected hypothesis (??). The goal should be discharged and the **Proof Control View** should now show the original goal  $c \in Q \setminus x$ . Click a second time on the **p0** button in order to finalize the proof. The smiley in the **Proof Control View** should now become green indicating that all sequents of the proof tree are discharged as shown in figure ??.

After saving the proof, the proof obligation `remove_1/inv2/INV` in the **Event-B Explorer** should also become a green chop.



There are alternative ways to prove the proof obligation. For instance we can use the **lasso** button to load into the **Selected Hypotheses View** those hidden hypotheses that contain identifiers in common with the goal, and with the selected hypotheses.

In order to move to the next undischarged proof obligation, you may also use the **Next Undischarged PO** button (The small right arrow icon) of the **Proof Control View** (3.40). The next proof can be solved the same way as the last one.



As an exercise, try to prove **Celebrity\_2**. A small hint: We have to fill in an existential quantifier. First, look in the list of hypothesis if you find any variable that is in **P**, and select that hypothesis. Then, instantiate **b'** and **R'** correctly (For instance, if you want to instantiate **b'** with **c**, then  $P \setminus \{c\}$  is a good choice for **R'**) by typing the instantiations in the **Goal View** (3.26) and then clicking on the red existential quantifier. Now, all open branches of the proof tree can be proved with **p0**. After this, we have completed all the proofs, and the model is ready for use.

This concludes the tutorial. Please note that this tutorial gave you only a first view of proving.

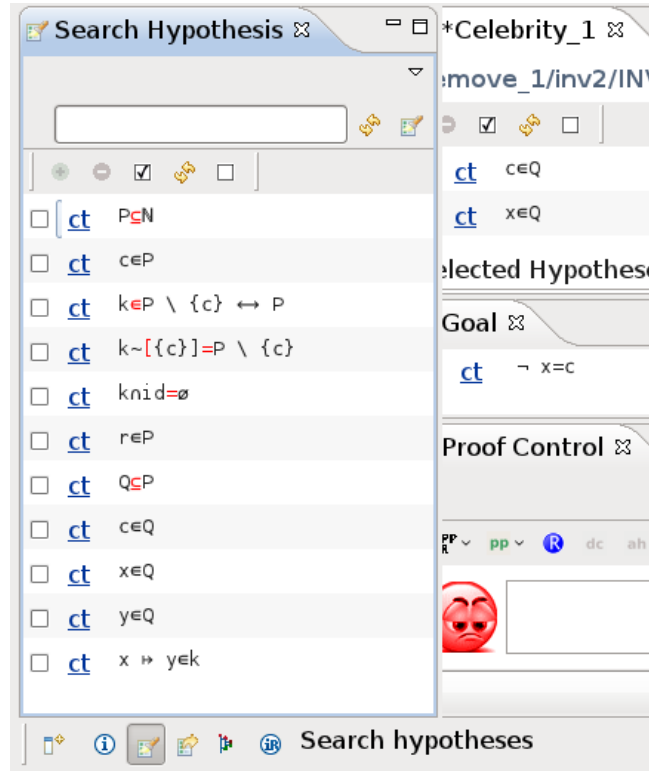


Figure 2.18: Search Hypothesis View

## 2.10 Location Access Controller



**Goals:** In this section, we will take a closer look at a few more complex proofs. For this, we will use the model of a Location Access Controller. The goal is to develop the proofs for deadlock freeness of the initial model and of the first refinement.

Through the model used in this section, we study a complete system and remind the proof rules of formal development. The system's task is to control the access of certain people to different locations of a site. The system is thus based on the authorization that a person has (or not has) access to a particular location.

Before the start to describe the initial model, import the archive file [Doors.zip](#) containing the model. For this select File › Import › General › Existing Project into Workspace. Then, select the according archive file and click on Finish. Rodin takes a few seconds to extract and load all the files.

### 2.10.1 Initial Model

Let us look at the initial model, consisting of `doors.ctx1` and `doors.0`. There are two carrier sets in the model context, one for people ( $P$ ) and one for locations ( $L$ ). Then, there is a location called outside (`outside`) and a relation (`aut`) which reflects where people are allowed to go. Everyone is allowed to go `outside`. The model machine has one event (`pass`), which changes the location of a person and one variable (`sit`), which denotes where a person is located.

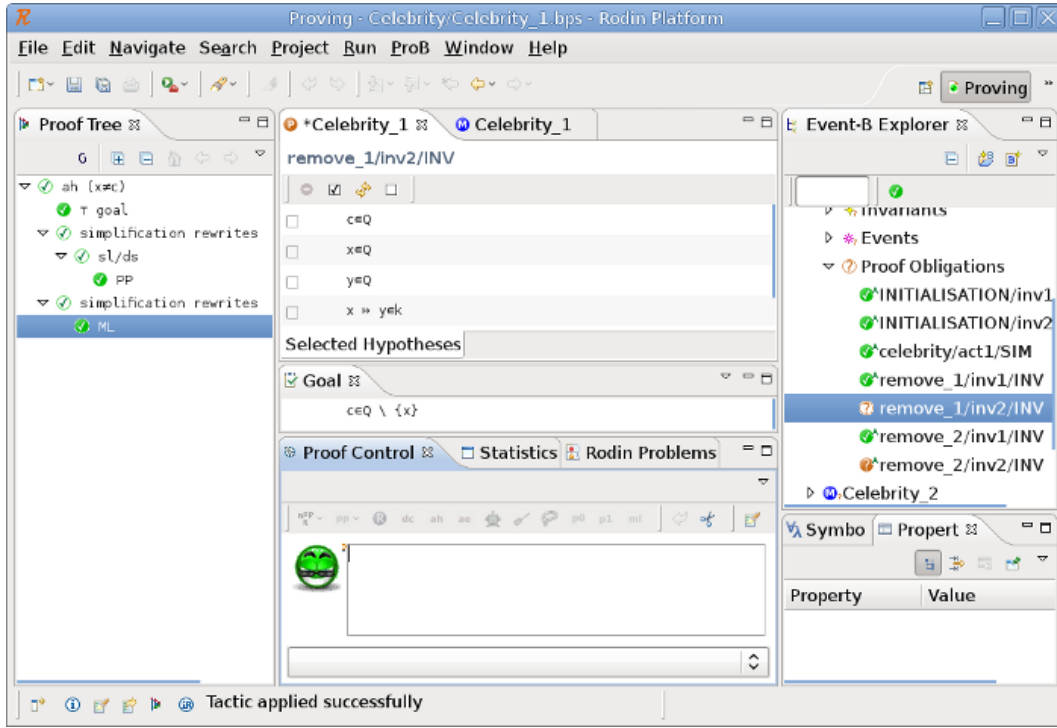


Figure 2.19: The green smiley indicates that all sequents of the proof tree are discharged

### Deadlock Freeness

Looking through the initial model, you will see that everything already has been proved. This is true, however, Rodin does not do any deadlock freeness (??) proof yet, so we will have to add them by ourself. A model is considered as deadlocked if the system reaches a state with no outgoing transitions. The objective of this section is to develop the proofs for deadlock freeness of the initial model and of the first refinement.

Consider the event `pass` of the initial model:

#### EVENTS

**Event** `pass`  $\hat{=}$

**any**

$p$   
 $l$

**where**

`grd11` :  $p \mapsto l \in \text{aut}$

`grd12` :  $\text{sit}(p) \neq l$

**then**

`act11` :  $\text{sit}(p) := l$

**end**

**END**

Since, the initial model has only one event `pass`, the system might deadlock when both guards of the event (`grd11` and `grd12`) are together false. Thus, proof of deadlock freeness means proving, that someone can always change room. Clearly, we want to avoid this happening. We have thus to prove that the two guards are always true. For this add a new derived invariant to `doors_0` called `DLF` (making it derived means clicking the `theorem` button) and change the predicate so that it is the conjunction (??) of the two guards:

## INVARIANTS

**DLF** :  $\exists p, l. (p \mapsto l \in \text{aut} \wedge \text{sit}(p) \neq l)$



Verweis auf ProB Plugin mit Deadlock Freeness Funktion

Save the machine. We will see that the autoprover (??) fails to prove the theorem DLF/THM.



If you don't see the proof obligation DLF/THM maybe you forgot to mark the invariant as a theorem by clicking on the **theorem** button. Another reason that you don't see the proof obligation DLF/THM could be that you forgot to rename the invariant as "DLF".

Let us analyze whether this is an inconsistency in the model. Switch to the **Proving Perspective** and double click on the proof obligation DLF/THM.

In order to succeed with the proof, we need a tuple  $p \mapsto l$  that is in **aut**, but not in **sit**. Searching the hypotheses, we find the **axm4** of **doors\_ctx1**, which states that there is a location  $l$ , where everyone is allowed to go:

## AXIOMS

**axm4** :  $\exists l. l \in L \setminus \{\text{outside}\} \wedge P \times \{l\} \subseteq \text{aut}$

So, for every person  $p$  in  $P$ ,  $p \mapsto l$  and  $p \mapsto \text{outside}$  is in **aut**. Since these are different, at least one of them cannot be in the function **sit**. Now, all we would need to prove is that  $P$  is nonempty. This holds, as carrier sets always are nonempty, but is a bit hard to derive.



In the Proof Control view, first disable the post-tactics (there is a dedicated button in this view on the top right corner, up to the toolbar).

Then add the hypothesis  $\exists x. x \in P$  using the **ah** button. Then, click on the Auto Prover button (The button with a robot on it). Other provers do not work here. After successfully adding the hypothesis, we can conclude the proof as follows:

Click on the existential quantifier of the expression  $\exists x. x \in P$  (appearing in the **Selected Hypothesis View**) as demonstrated in figure 2.20. You see that it is automatically instantiated, it leads to the selected hypothesis  $x \in P$ . We can now instantiate  $p$  in the goal with  $x$ : enter  $x$  in the yellow box corresponding to  $p$  in the **Goal View** and click on the existential quantifier as shown in figure 2.21.

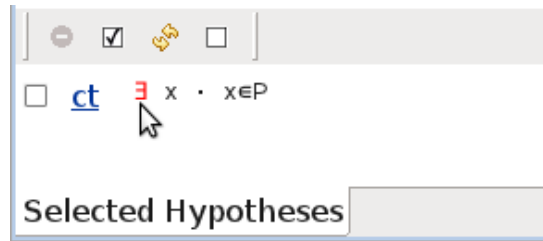


Figure 2.20: Click on the existential quantifier in order to ...

It remains now to instantiate  $l$ . In order to instantiate it, we need a case distinction. Type  $\text{sit}(x) = l$  into the **Proof Control View** and click on **Case Distinction** button (**dc**) to look at the two cases  $\text{sit}(x) = l$  and  $\text{sit}(x) \neq l$ . Before starting with the cases, the prover now wants you to prove that  $x \in \text{dom}(\text{sit})$ . This can be done with the **p0** prover, as **sit** is a total function (??). In the first case  $x$  is situated in  $l$ , so it cannot be in **outside**. So, you can instantiate  $l$  with **outside** (type **outside** in the box corresponding to  $l$  and click on the existential quantifier). In order to prove that  $x$  is allowed to **outside**, you will need to select the hypothesis

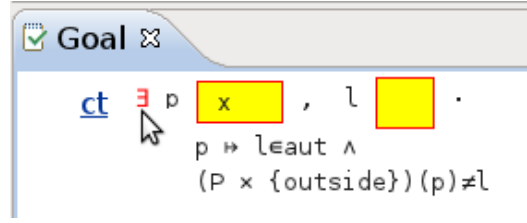


Figure 2.21: TODO: Description

$P \times \text{outside} \subseteq \text{aut}$  (if this hypothesis doesn't appear in the search hypothesis, type `outside` in the proof control view and click on the **Search Hypothesis** button, and add it to the selected hypothesis with the green plus icon next to it). Then you can finish this case with the `p0` prover. If you look at the proof tree, you see that you now have reached the other branch of the case distinction. In this case, you can simply instantiate `l` with `l`, as `x` is not situated there. Finally, click on the `p0` button to finish the proof.

### 2.10.2 First Refinement

Now we are going to detail the main complexity of our model: the deadlock freeness proof for the first refinement.

The difference between the first refinement and the initial model is that a new constant `com` been added in order to describe which rooms are connected. Additionally, we have a constant `exit`, which will be explained later.

Concerning the events, **INITIALISATION** does not change, whereas the event **PASS** is refined as a consequence. Indeed we estimate that a person can move to another location `l` if there is the authorization to be in `l` (already defined in the abstraction) and also location `l` communicates with the location where `p` is at this precise moment (represented by `sit(p)`).

Consequently, the guard of this new event version is better than the precedent, since we have :

$$(\text{sit}(p) \mapsto l \in \text{com}) \Rightarrow (\text{sit}(p) \neq l)$$

In fact, we are obviously faced with a difficulty here; it is not possible to prove that the refined event **PASS** does not happen less often than its more abstract homologue. To demonstrate this, we would have to prove that the guard of the abstract event implies that of the concrete event.

The issue is that this condition cannot be verified in general. Moreover the failure to prove the above condition indicates that there are possibilities that certain people could stay permanently blocked in locations. Besides the geometry of communication between locations clearly introduces an additional constraint limiting the way people can move.

As a consequence our study has revealed a problem in our method that was obviously ignored in the document requirements. So we must find a sufficient solution.



Please note that the post-tactics should still be disabled before starting this part of the tutorial.

At the beginning of this section we need to come back to the **Event-B Perspective**. Like in section 2.10.1 open `door_1` machine and add a derived invariant (theorem) called **DLF** as follows:

#### INVARIANTS

$$\text{DLF} : \exists q, m \cdot (q \mapsto m \in \text{aut} \wedge \text{sit}(q) \mapsto m \in \text{com})$$

Save the file. Once again, the prover fails to prove the deadlock freeness automatically. Actually all we want is that "any person authorized to be in a location must also be authorized to go in another location which communicates with the first one".

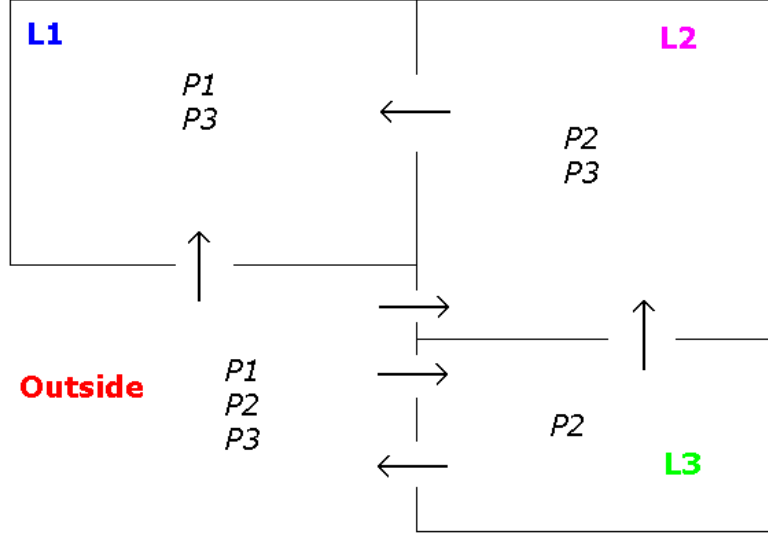


Figure 2.22: TODO: Description

This can be illustrated as demonstrated in figure 2.22.

At the beginning of the proof, there are no selected hypothesis at all. So we need to select a few. The old deadlock freeness invariant will be useful, `axm7` of `doors_ctx2` too.

### AXIOMS

$$\text{axm7} : \exists l. l \in L \setminus \{\text{outside}\} \wedge \text{outside} \mapsto l \in \text{com} \wedge P \times \{l\} \subseteq \text{aut}$$

To begin with, we try to avoid using `exit`, as we want to keep things as simple as possible. But since `sit` and `aut` are inside the invariant, we also are likely to need

$$\text{sit} \subseteq \text{aut} \text{ and } \text{sit} \in P \rightarrow L.$$

Once again, the prover automatically rewrites the existential quantifiers in the hypotheses. We now look at the proof. There is an easy case if  $\text{sit}(p) = \text{outside}$ . Add this case as previously using the **Case Distinction** button (`dc`) and solve it. For `q`, the choice `p` is obvious. For `m`, you will use the existential quantifier of `axm7` of `doors_ctx2` to instantiate `m` with `l` as `l0`.

For the other case, we will need the notion of `exit`. This function `exit` connects locations to locations and defines at every location except `outside`.

We can look at the axioms about `exit`:

### AXIOMS

$$\text{axm3} : \text{exit} \in L \setminus \{\text{outside}\} \rightarrow L$$

$$\text{axm4} : \text{exit} \subseteq \text{com}$$

$$\text{axm5} : \forall s. s \subseteq \text{exit}^{-1}[s] \Rightarrow s = \emptyset$$

$$\text{axm6} : \text{aut} \triangleright \{\text{outside}\} \subseteq (\text{aut}; \text{exit}^{-1})$$

The axioms state that:

- (axm3) Every room except the outside has exactly one exit.
- (axm4) An exit must be a room that communicates with the current one
- (axm5) A chain of exits leads to the outside without any cycles or infinite paths
- (axm6) Everyone allowed in a room is allowed to go through its exit.

In our proof, we still need to show that anyone who is not **outside** can walk through a door. For this, **axm5** is useless, so we add all hypotheses containing **exit** except for **axm5**. Now we have to instantiate **q** and **m** correctly and to conclude that the proof should not be too hard. Once again, for **q**, the choice **p** is obvious. But it is not quite as easy for **m**. Expressed in language, **m** must be the room behind the exit door of the room that **p** is currently in.



Try translating this into set theory. But do not worry if you get it wrong. You can still go back in the proof by right-clicking at the desired point in the proof tree and choosing **Prune** in order to retry.

## 2.11 Outlook

Congratulations – if you made it this far, you should have a good foundation for getting some real work done with Rodin. In the following we’d like to provide you with a few pointers that will help you to make your work as efficient as possible.

**Use the Reference Section and FAQ** If you have a specific issue or if you quickly need to look something up, check the reference (3) and FAQ (4) of this handbook.

**There is an Online, PDF and an Eclipse-Version of this Handbook** There are three versions of this handbook. You can access it directly through Rodin by using the built-in help browser (**Help** > **Help Contents**). In particular, the Eclipse-Version can be used offline.

**Use the Rodin Wiki** The Rodin Wiki (3.45) contains the latest news regarding Rodin and a wealth of information that is not in the scope of this handbook. Be sure to check out what’s there.

**Find useful Plugins** There are many plugins available, be sure to check them out. There is a good chance that they will make your life easier.

**Subscribe to the mailinglists** The wiki documents the existing mailinglists, which include a list for users and evelopers. We strongly recommend to subscribe to the announcement list.

**Rodin in Industry** If you consider using Rodin in Industry, be sure to explore the deliverables from the Deploy (3.16) project, where industrial partners describe their experiences with Rodin.

We wish you success in your modeling projects!



# Chapter 3

## Reference

### 3.1 Outline

1. The Rodin Platform: This chapter should give the user an overview about the UI elements he might encounter.
  - (a) Eclipse in general: just references to other resources
  - (b) The Event-B perspective: We briefly describe the various views (Event-B explorer, structure editor, outline view, symbol view) and their content. We want to cover every element of each view (like buttons, entry fields, etc.), but for the underlying theory we link to the other reference sections. The relevant menu entries are also described.
  - (c) The proving perspective: We do basically the same as for the Event-B perspective. The views are proof tree, goal, control and statistics.
  - (d) Preferences: We briefly describe what preference can be set for Rodin. For their deeper meaning, we'll refer to the other sections (esp. the prover settings).
2. Event-B (modeling notation)
  - (a) Contexts
    - i. Sets: Introducing sets, with a comment about how to add enumerated sets
    - ii. Constants and axioms (with theorems), which POs are generated
    - iii. Extending a context
  - (b) Models
    - i. Variables and invariants (with theorems), generated POs
    - ii. Seeing a context
    - iii. Refining a model: Refines clause, gluing invariant, general picture of refinement with pointers to the event below, re-use of variables
    - iv. Events: All aspects of events are covered: Parameters, guards and actions, generated POs, refinement (again with generated POs, re-use of parameters, witnesses), status (normal/convergent/anticipated), merging of events.
    - v. Termination: How to prove termination by the model's variant and the status of events
  - (c) Generated proof obligations: We give a brief overview about what POs are generated where. This should help the user to identify the reason of a PO when he just know its label.

3. Mathematical notation: Here we cover the complete mathematical notation of Event-B. For each expression (like an operator) we describe its purpose, its type, the type of its arguments and well-definedness-conditions. We roughly separate the expressions into three groups: predicates, set-theoretical and arithmetic.

Some laws (like the commutative law of addition in arithmetic) would be nice but will not be part of the first iteration of the documentation. E.g. the Z reference manual does this.

References to related proof rules would be nice to have, too. But again, this is not part of the first iteration.

- (a) Introduction: What data types exist, what are well-definedness-conditions, how the description of the expressions is organized.
  - (b) Predicates: All operators that work with predicates ( $\wedge$ ,  $\vee$ , quantifier, ...).
  - (c) Sets and relations
  - (d) Arithmetic
4. Proving: This section should enable the user to understand proofs in Event-B. In the first iteration, we won't go into detail when it comes to proof rules and tactics. We just refer to the already existing Wiki sites.
  - (a) Introduction: How does a proof in Event-B look like? (Goal, proof tree, hypothesis, selected hypothesis, ...)
  - (b) Proof rules (first only a link to the Wiki page)
  - (c) Proof tactics (first only a link to the Wiki page)
  - (d) The provers (PP, ML, New-PP): We just give a very brief over the existing provers. (What is its strength/weakness).
5. Glossary: During planning the outline of the reference we encountered some topics that did not seem to fit anywhere. This might be a good place to put such things. (E.g. horizontal vs. vertical refinement, naming convention, ...)
6. Index (optional): An index is very helpful to find the relevant information in the documentation. We make the index optional because there is the alternative of a text-based search (esp. in the web version of the documentation) and we ran into technical difficulties when generating the index for the web version.

## 3.2 Abstract Machine

## 3.3 Abstract Machine Notation

## 3.4 Arithmetic

## 3.5 Atelier B Provers

## 3.6 Auto Prover

## 3.7 Axiom

## 3.8 B

A formal modeling notation. Two dialects are classical B (3.10) and Event-B (??).

## 3.9 Camille

Camille is an alternative, text-based editor. It can be installed through the Eclipse Install mechanism. More information is available in the Rodin Wiki (3.45).

## 3.10 Classical B

## 3.11 Constant

## 3.12 Context

## 3.13 Data Refinement

## 3.14 Datatypes

List of Event-B datatypes

## 3.15 Deadlock

## 3.16 Deploy

The Deploy Project funded the Rodin project in part. More information at <http://www.deploy-project.eu/>.

## 3.17 Eclipse

- Eclipse Definition
- Pointers to Web Tutorials, etc.

## 3.18 Editor View

## 3.19 Event

Definition event

## 3.20 Event-B

Event-B is a formal method (3.24) for system-level modelling and analysis. Key features of Event-B are the use of set theory (3.61) as a modelling notation, the use of refinement (3.44) to represent systems at different abstraction levels and the use of mathematical proof to verify consistency between refinement levels.

See Also:

- <http://www.event-b.org>

## 3.21 Event-B Component

Machines (3.31) and Contexts (3.12) are components.

## 3.22 Event-B Explorer

The View showing the Event-B projects and their content. In the default Event-B perspective, it is the slim browser on the left edge of the Workspace. If it is missing, make sure that you use the correct perspective. You can explicitly enable it with `Windows › Show View... › Event-B Explorer`.

## 3.23 First Order Predicate Calculus

## 3.24 Formal Method

## 3.25 Gluing Invariant

## 3.26 Goal View

## 3.27 Guard

## 3.28 IDE

Integrated Development Environment

## 3.29 Initialization

Every machine has a special event `INITIALIZATION` that will be used to initialize the machine's state.  
TODO: Determinism, refinement.

## 3.30 Label

## 3.31 Machine

## 3.32 Mathematical Notation

## 3.33 Menu Bar

## 3.34 Model Checker

## 3.35 Naming Convention

In this section we describe a recommended naming convention. Good naming conventions save time – and nerves.

### **3.36 Non-Deterministic**

### **3.37 Outline View**

### **3.38 P0 Prover**

### **3.39 Partition**

### **3.40 Proof Control View**

### **3.41 Proof Obligation**

### **3.42 Proof Tree View**

### **3.43 Plugin**

### **3.44 Refinement**

**Horizontal Refinement**

**Vertical Refinement**

**Data Refinement**

See also:

- Data refinement in the traffilight tutorial ([2.8.1](#))

### **3.45 Rodin Wiki**

<http://wiki.event-b.org/>

### 3.46 Structural Editor

### 3.47 Symbols View

### 3.48 Predicate Logic

### 3.49 ProB

### 3.50 Project

### 3.51 Proof Obligation Labels

### 3.52 Propositional Calculus

### 3.53 Relation

### 3.54 Rodin

... Rodin Definition ...

### 3.55 Rodin Download

Download Page: [http://sourceforge.net/projects/rodin-b-sharp/files/Core\\_Rodin\\_Platform/](http://sourceforge.net/projects/rodin-b-sharp/files/Core_Rodin_Platform/)

### 3.56 Rodin Platform

### 3.57 Rodin Problems View

### 3.58 Rodin Nature

Eclipse Projects can have one or more natures to describe their purpose. The GUI can then adapt to their nature. Rodin Projects must have the Rodin-Nature. If you create an Event-B project, it automatically has the right nature. If you want to modify an existing project, you can edit the `.project` file and add the following XML in the `<natures>` section:



```
<nature>org.rodinp.core.rodinnature</nature>
```

### 3.59 Sees

### 3.60 Set

### 3.61 Set Theory

... Set Theory Definition ...

**3.62 Temporal Logic**

**3.63 Theorem**

**3.64 Tool Bar**

**3.65 Undischarged Proof Obligations**

**3.66 Upgrade**

**3.67 Witness**

**3.68 Wizard**

**3.69 Zip File**



## Chapter 4

# Frequently Asked Questions

### 4.1 General Questions

#### 4.1.1 What is Event-B?

Event-B is a formal method for system-level modelling and analysis. Key features of event-B are the use of set theory as a modelling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to verify consistency between refinement levels. More details are available in <http://www.event-b.org>.

#### 4.1.2 What is the difference between Event-B and the B method?

Event-B (3.20) is derived from the B method. Both notations have the same inventor, and share many common concepts (set-theory, refinement, proof obligations, ...) However, they are used for quite different purpose. The B method is devoted to the development of correct by construction software, while the purpose of event-B is to model full systems (including hardware, software and environment of operation).

Both notations use a mathematical language which are quite close but do not match exactly (in particular, operator precedences are different).

#### 4.1.3 What is Rodin?

The Rodin Platform is an Eclipse-based IDE for Event-B that provides effective support for refinement and mathematical proof. The platform is open source, contributes to the Eclipse framework and is further extendable with plugins.

#### 4.1.4 Where does the Rodin name come from?

The Rodin Platform (3.56) was initially developed within the European Commission funded Rodin project (IST-511599 ), where Rodin is an acronym for "Rigorous Open Development Environment for Complex Systems". Rodin is also the name of a famous French sculptor, one of his most famous work being the Thinker.

## 4.2 Installation Questions

## 4.3 Proofer Questions

## 4.4 Usage Questions

### 4.4.1 Where vs. When: What's going on?

## 4.5 Plug-In Questions